Forord

Denne rapporten er resultatet av hovedoppgaven i 5. års-kurs ved Norges teknisk-naturvitenskapelige universitet (NTNU), institutt for datateknikk. Vi har vært to studenter på denne oppgaven som begge har bakgrunn fra høgskole (Høgskolen i Buskerud og Norges Informasjonsteknologiske Høgskole i Oslo) før vi begynte på NTNU i 4. års-kurs høsten 2003.

Utgangspunktet for problemstillingene bygger på vår erfaring og interesse med arbeidet på fordypningsprosjektet høsten 2004, samt ideer fra SnapTV som tok kontakt med oss for et samarbeid om oppgaven.

Vi vil gjerne takke Jon Bøhmer og Marius Kjeldal i SnapTV for presentasjon av problemstillinger og for å komme med innspill underveis i prosjektarbeidet.

Til slutt vil vi takke Roger Midtstraum som har vært faglig veileder og stipendiat Gisle Grimen som har stilt seg til rådighet for oss når vi har hatt spørsmål og diskusjon rundt fremgangsmåte på noen områder.

Trondheim 13. juni 2005

Christian Andersen

Lars Moe

Sammendrag

Denne rapporten beskriver en metode og et prototypesystem for beskyttelse av IPTV. Utgangspunktet for oppgaven er et konkret behov for beskyttelse i et eksisterende IPTV-system. Behovet er basert på krav fra rettighetshaverne om tilstrekkelig beskyttelse av deres intellektuelle eiendom (i form av filmer, TV-serier og -programmer) mot uautorisert tilgang og kopiering.

IPTV er en fellesbetegnelse på teknologi for å overføre digitale TV-signaler ved hjelp av IP-protokollen. En spesialbygd datamaskin (set-top-boks) sørger for å ta i mot disse signalene hos TV-seeren, og omformer de signalet slik at man kan benytte et vanlig TV-apparat uten støtte for mottak av digitale TV-signaler.

Ved hjelp av en prototype viser vi en transparent metode for ende-til-ende beskyttelse av IPTV. Løsningen er uavhengig av videoavspiller, media-server og innholdsformat.

Den innholdsagnostiske egenskapen ved løsningen begrenser muligheten for å benytte spesialiserte metoder for videokryptering. Vi baserer oss derfor på den standardiserte krypteringsalgoritmen AES i CTR-modus, med mulighet for å velge alternative algoritmer. Det foreslås også en enkel selektiv krypteringsalgoritme for å begrense mengden av videodata som krypteres. Disse krypteringsmetodene tar hensyn til behovet for distribusjon av videodata med unicast- og multicast.

Løsningen benytter sanntidsbytte av krypteringsnøkler. Denne nøkkeldistribusjonen er implementert ved å bruke en sikker unicast-kanal mellom klientapplikasjonen og serveren.

Innhold

Figurer

Tabeller

Del I

Problemområdet

Kapittel 1

Introduksjon

1.1 Bakgrunn

Med større utbredelse av bredbånd og installasjon av raske nettverkslinjer, blir IPTV[1] en mer og mer aktuell teknologi. Dette er en del av en prosess hvor IP-protokollen blir brukt til å tilby tjenester som e-post, web, telefoni, musikk/film-nedlasting og TV-tjenester.

I de siste årene har det vært mye fokus på ulovlig nedlasting av musikk og film, og det er naturlig for rettighetshaverne å se seg om etter løsninger som kan begrense ulovlig tilgang og videreformidling av deres innhold. Dette gjøres for å beskytte rettighetshavernes interesser, og sikre at beskyttede sendinger kun kan aksesseres med gyldig abonnement. For å oppnå dette vil rettighetshaverne kreve funksjonalitet for digital rettighetsstyring (DRM) i systemet. Hvis vi sammenligner IPTV med dagens digitale kabel- og satellittsendinger, vil DRM være en mer avansert form for Conditional Access (CA), som disse opererer med i dag.

1.2 Hensikt

Oppgaven er i hovedsak basert på en henvendelse fra selskapet SnapTV (se avsnitt 2.3), en leverandør av en IPTV-løsning. Som første skritt på veien ønsket de en løsning som kan beskytte TV-sendingene de distribuerer i sitt system mot ulovlig tilgang. Et viktig kriteria var at beskyttelsen bør være mest mulig fleksibel i forhold til formatet på mediastrømmen. SnapTV ønsket heller ikke en løsning som er avhengig av en bestemt set-top-boks eller avspiller. Hensikten vår er å prøve ut hvordan dette kan la seg implementere, og undersøke problemstillinger rundt en slik løsning.

[1] TV-signaler over IP-protokollen frem til forbrukeren. Signalene blir dekodet av en set-top-boks i tilknytning til TVen.

1.3 Problemstilling

Vår oppgave vil være å undersøke hvordan digital rettighetsstyring (DRM) kan implementeres i et IPTV-system. Systemet tilbyr blant annet multicast-sendinger, video-on-demand (VoD) og tidsforsinkelse på TV-sendinger.

Sikkerhet og beskyttelse bør ideelt sett tilrettelegges fra grunnen av i et system. Vi vil se på hvilke problemer som kan oppstå ved å bygge DRM-funksjonalitet rundt et eksisterende system. Fokus vil være på de mest operasjonkritiske elementene i et DRM-system.

En del av problemstillingene krever at det implementeres en prototype av systemet som inneholder nødvendig funksjonalitet for å besvares. Denne prototypen må utvikles med tanke på at den skal kunne utvides og gjøres robust nok til å kunne brukes i et eksisterende IPTV-system.

1.3.1 Del-problemstillinger

- Transparent beskyttelse på eksisterende strømmer. Dette er relevant da standarder for mediaformater ofte er svært nye og kan endre seg med få års mellomrom. Det vil være en fordel om beskyttelsesmekanismene kan gjøres uavhengig av klient- og serversystem. For eksempel ved å bruke protokolluavhengige metoder for kryptering av nettverkskommunikasjon.

  Problemstilling

  - Kan dette gjøres sømløst og uavhengig av videoavspiller, serversystem og codec?

- Beskyttelse av multicast-protokoller for streaming og on-demand beskyttelse av VoD-strømmer. Datastrømmene er i utgangspunktet ubeskyttede. For å oppnå ønsket grad av beskyttelse må disse strømmene krypteres. Systemet vil ha bruk for forskjellige typer datastrømmer (TCP/UDP/multicast). Dette må tas hensyn til i metoden man bruker for å beskytte strømmene.

  Problemstillinger

  - Hvordan kan forbindelsesløse multicast-strømmer krypteres?
  - Undersøke behov for bruk av ulike metoder for kryptering av multicast- og unicast-strømmer.

- Undersøke behov og metoder for nøkkelbytte- og distribusjon. Spesielt med hensyn til gruppenøkler ved bruk i beskyttelse av multicast-strømmer. Nøkkelskifte vil være nødvendig for multicast-strømmer for å hindre at uvedkommende kan (over tid) finne ut nøkkelen. Multicast-strømmer vil kreve lik nøkkel hos alle abonnentene. Ved bortfall av en abonnent må nøkkelen kunne oppdateres hos gjenværende abonnenter.

Problemstilling

- Utvikle en metode for distribusjon av krypteringsnøkler for krypterte multicast- og unicast-strømmer.

- Ressursbruk og -krav for beskyttelsen. Klientsiden av systemet vil kunne bestå av enheter med begrensede ressurser. Dekryptering av strømmene vil ta ekstra ressurser i tillegg til dekoding og avspilling av innholdet. Serversiden av systemet vil ha mange klienter tilkoblet og man må kunne finne ut hvor mange klienter en server kan betjene.

  Med tanke på klienter som har begrensede ressurser vil det kunne være nødvendig å vurdere enklere krypteringsalgoritmer. Krypteringsnøkkelen må byttes ut hvis det er en mulighet for at algoritmen kan knekkes i løpet av rimelig kort tid.

  Problemstilling

  - Undersøke ressursbruken ved aktuelle beskyttelsesmekanismer.

- Prototype

  - Prototypen vil hjelpe oss til å finne konkrete problemer ved implementasjon av beskyttelse for IPTV.
  - Prototypen vil gjøre det mulig for oss å måle ytelse.

1.4 Avgrensing av problemstilling

Prototypen vil være en enkel løsning for å konkretisere noen av ideene våre. Det mest nødvendige av funksjonaliteten for å få undersøkt og belyst viktige punkter vil bli implementert. Dette innebærer å forenkle implementasjonen på noen områder.

Prototypesystemet vil ikke ta høyde for uredelige brukere, eller på noen måte gardere mot dette i form av for eksempel kryptert intern nøkkelbehandling, beskyttet minne, eller vannmerking av innhold.

Det vil bli holdt en teknisk vinkling på DRM, og juridiske områder som patentregler, personvern og opphavsrett blir ikke belyst, selv om dette er hovedgrunnene bak beskyttelseskravene til IPTV-løsningene.

1.5 Resultater

Ved hjelp av en prototype viser vi en transparent metode for ende-til-ende beskyttelse av IPTV. Løsningen er uavhengig av videoavspiller, mediaserver og innholdsformat.

Den innholdsagnostiske egenskapen ved løsningen begrenser muligheten for å benytte spesialiserte metoder for videokryptering. Vi baserer oss derfor

på den standardiserte krypteringsalgoritmen AES i CTR-modus, med mulighet for å velge alternative algoritmer. Det foreslås også en enkel selektiv krypteringsalgoritme for å begrense mengden av videodata som krypteres. Disse krypteringsmetodene tar hensyn til behovet for distribusjon av videodata med unicast- og multicast.

Løsningen benytter sanntidsbytte av krypteringsnøkler. Denne nøkkeldistribusjonen er implementert ved å bruke en sikker unicast-kanal mellom klientapplikasjonen og serveren.

1.6 Organisering av rapporten

Rapporten består av fem deler. Første del beskriver problemområdet, inkludert problembeskrivelse og en innføring i viktige begreper brukt i rapporten.

Del nummer to beskriver krav og overordnet design av en proof-of-concept implementering (prototype) for transparent IPTV-beskyttelse.

Detaljer om hvordan prototypen er implementert, hvilke programmer og biblioteker som er brukt for å støtte utviklingen finnes i del tre. Denne delen avsluttes med et testkapittel hvor resultater fra eksperimentering med prototypen presenteres..

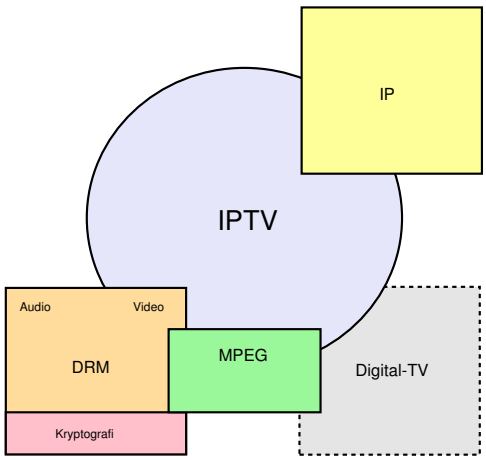
I del fire vil vi drøfte løsningen opp mot problemstillingene og beskrive de erfaringene vi har gjort under arbeidet.

Til slutt i rapporten er det lagt med en del tilleggsdokumentasjon. Dette dokumenterer mer av arbeidet som er gjort og støtter opp rapporten med klassediagrammer, eksprimentresultater, utdrag fra kildekode og brukerveiledning for programmene. Til rapporten skal det også følge med en CD-ROM som beskrevet i vedlegg G.

Kapittel 2

Bakgrunn

Dette kapittelet vil gå gjennom enkelte begreper og teknologier som er viktig for gjennomføringen og forståelsen av oppgaven. På figur 2.1 er IPTV satt inn i kontekst med beslektede teknologier. Her er sammenhengen mellom IPTV og DRM (digital rettighetsstyring) sentral, hvor DRM i stor grad baserer beskyttelsesmekanismer på kryptografi. Samtidig er det sanntidsoverføring av data over IP-nettverk som er i fokus, i motsetning til nedlastbare statiske filer. Her kommer MPEG inn som det viktigste distribusjonsformatet. Digital-TV[1] er med på figuren for å illustrere slektskapet dette har med IPTV. Etter hvert som IPTV blir mer utbredt skal man ikke se bort i fra at man vi se en konvergens mellom disse to teknologiene.

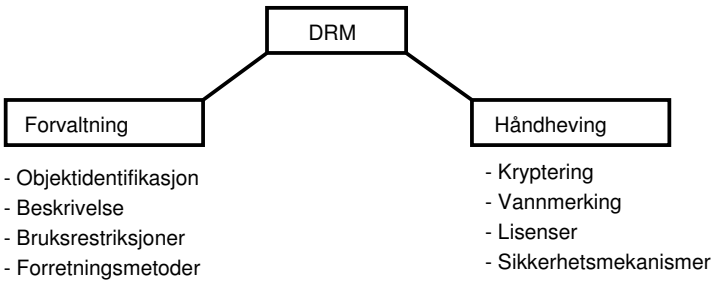


Figur 2.1: IPTV i teknologisk sammenheng

[1]Med digital-TV menes levering av nåværende kabel- og satellittsystemer som transporterer MPEG-2-signaler i henhold til DVB-standardene (Digital Video Broadcasting).

2.1 Digital rettighetsstyring

Digital rettighetsstyring (DRM) handler om å distribuere, kontrollere og begrense tilgang og bruk av digitale verk (dokumenter, audio, video, bilder). Rapporten [AM04] gir en generell beskrivelse av DRM. Her kommer det frem en todeling av DRM som vist i figur 2.2. I [AM04] konsentrerte vi oss mest



Figur 2.2: Todeling av DRM [AM04]

om forvaltningsdelen, som innebærer identifisering, beskrivelse (metadata), bestemme rettigheter og avgjøre ønskede salgs- og distribusjonsmodeller for det digitale innholdet. Denne oppgaven omhandler i stor grad problemstillinger i håndhevingsdelen, som går ut på å sikre at bruksrettighetene for det distribuerte innholdet overholdes.

Det er flere elementer for beskyttelse av innhold i et komplett DRM-system:

- Kopibeskyttelse
- Sporing av ulovlig eksemplarfremstilling (for eksempel vannmerking)
- Ende-til-ende beskyttelse (distribusjons-sikkerhet)

Vi har i rapporten konsentrert oss om ende-til-ende beskyttelse i form av sanntidskryptering av media. Og så lenge klientapplikasjonen ikke har mulighet til å lagre videodata lokalt, eller kopiere dette ut av systemet, vil behovet for kopibeskyttelse og kopisporing ikke være til stede.

Hvis det ønskes en mer generell innføring og bakgrunnsinformasjon om DRM, henviser vi leseren til [AM04].

2.1.1 Lisens

For å gi en sluttbruker lov til å spille av innhold må denne ha en gyldig lisens. En lisens kan sies å være beskrivelse av hva brukeren har lov til å gjøre med et gitt innhold (musikk, video eller tekst). En typisk DRM-lisens, i et system hvor man for eksempel laster ned en lyd-fil, vil være noe du laster ned sammen med dataene og som kontrolleres av avspilleren hver gang du

bruker dataene. En lisens vil da tildeles etter at brukeren er godkjent som kunde (autentisert) og eventuelt har betalt for tilgang (autorisert). Lisensen kan inneholde en nøkkel som brukes til å dekryptere innholdet. Hvis ikke lisensen inneholder en slik dekrypteringsnøkkel vil den kunne fungere som en slags billett som gir tilgang til den nødvendige dekrypteringsnøkkelen.

Vi kan kalle en lisens man laster ned sammen med innholdet for en *offline*-lisens. Beskyttet innhold kan da aksesseres uten at man trenger tilgang til en sentral server. Det motsatte vil være en *online*-lisens hvor lisensdataene er lagret og sjekket på en sentral server. I dette tilfellet må man være tilkoblet denne serveren for å kunne spille av innholdet.

I et IPTV-system er dataene brukerne vil ha tilgang til i konstant forandring, og systemet er avhengig av å være online til en hver tid for å fungere. Dette gjør det mulig å flytte lisensoppbevaringen til serveren. Det eneste som trengs på klienten er informasjon nok til at serveren kan stole på klienten (se 2.7.4), for så å få oversendt en dekrypteringsnøkkel.

De fleste DRM-systemer bruker en eller annen form for XML-basert format til å beskrive lisensen, men i online-scenariet er det mulig å benytte en database eller andre måter å lagre informasjonen på. Online-lisensene vil beskrive det samme som en offline-lisens. Derfor er det viktig at lisensinformasjonen i en database, eller lignende på server, stemmer overens med en standard for rettighetsspråk. Dette vil gjøre det enklere for systemet å integreres med for eksempel betalingssystemer som kan stå for genereringen av lisensinformasjonen.

MPEG-REL

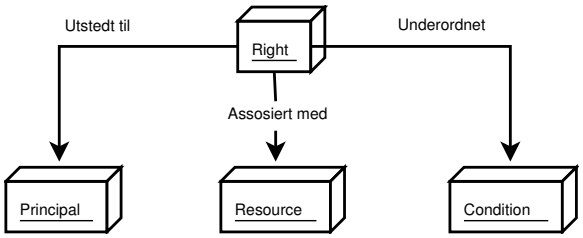
MPEG-REL er et rettighetsspråk som kan brukes som beskrevet ovenfor. Selv om det primært er laget for å være en offline-lisens, kan feltene og informasjonsinndelingen brukes i en online-lisens. Utviklingen av MPEG-REL startet i 2001, språket er basert på XrML [BH].

Modellen for rettighetsspråket MPEG-REL består av 4 deler som er definert av MPEG-REL under *grant* statements [Con]. En MPEG-REL *grant* består av følgende punkter [BH]: (se også figur 2.3)

- Principal. Til hvem *grant* er utstedet til.
- Right. Rettighetene *grant* spesifiserer.
- Resource. Ressursen som *grant* omfatter.
- Condition. Faktorene som må være møtt før rettighetene brukes.

2.1.2 Objektidentifisering

Digitalt innhold kan formateres og adressere på mange forskjellige måter, dette gjør at det har blitt utviklet flere måter å identifisere data på. Metadata



Figur 2.3: MPEG-REL

blir ofte brukt til å beskrive selve dataene, men skal man søke eller finne fram til data er det en fordel å ha en unik identifikator til alle datatyper. Noen standardiserte eksempler på slike unike identifiseringer er listet i tabell 2.1 [FK04]. I tillegg er det verdt å nevne MPEG-21 standarden, som er kort beskrevet i avsnitt 2.5.4, som også tar for seg identifisering av digitale objekter.

ID-kode	Forklaring	Innholdstype
BICI	Book Item and Component identifier	Bøker
DOI	Digital Object Identifier	Hva som helst
URI	Uniform Resource Identifier	Hva som helst
ISBN	International Standard Book Number	Bok
ISAN	International Standard Audiovisual Number	Audiovisuelle programmer
UMID	Unique Material Identifier	Audiovisuelt innhold

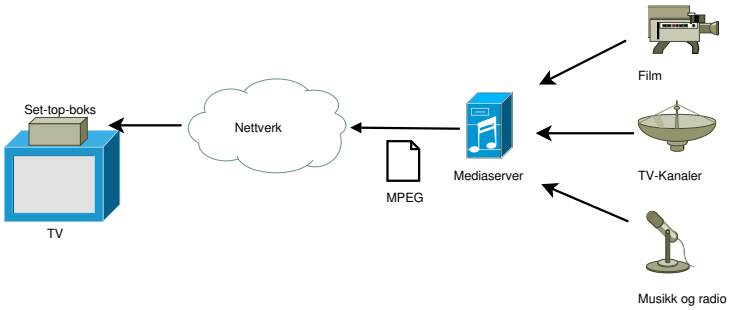
Tabell 2.1: Vanlige måter å identifisere innhold

2.2 IPTV

Internet Protocol Television (IPTV) har blitt en fellesbetegnelse på overføring av fjernsynssignaler og videosignaler over IP (Internet Protocol). IPTV begrepet favner sanntids-TV ved bruk av multicasting og valgt visning av enkeltprogrammer/video video-on-demand (VoD) (se figur 2.4). IPTV åpner også større muligheter for toveiskommunikasjon mellom brukeren og tjenesten. Dette muliggjør former for interaktiv TV (iTV), VoD, og et tilbud tilpasset den enkelte bruker av systemet [Wika].

IPTV åpner også for muligheter for andre avanserte tjenester, som for eksempel [Sch04]:

- Bilde-i-bilde (picture-in-picture)



Figur 2.4: IPTV-distribusjon

- Sammensetninger av programpakker som kan kjøpes, med utgangspunkt i interessefelt. For eksempel fotballpakker, filmer, sportsbegivenheter.
- Jukeboks mulighet.
- EPG, elektronisk programguide
- Tidsskift (timeskift)
- Informasjonsoppslag og tilleggsinformasjon
- Interaktive muligheter (for eksempel avstemning, quiz og opinionsundersøkelser)

2.2.1 Video-on-Demand

Med VoD har sluttbruker full kontroll over funksjoner som pause, spoling fremover og bakover. Brukeren har da mulighet å skifte kanal, og spole frem og tilbake i programmene etter eget ønske [Wikd].

Ved sanntids-TV vil naturlig nok ikke spoling fremover være mulig, men hvis du stopper eller pauser programmet vil du gå over til video-on-demand, og få din egen dedikerte TV-strøm igjen og ikke lenger se *live* multicast. Serveren vil da ta opp sendingen for deg, og brukeren kan se videre fra der vedkommende valgte å pause eller stoppe programmet.

2.2.2 Set-top-boks

For å koble til en IPTV-tjeneste må sluttbrukeren ha en enhet som kan kommunisere med IPTV-tjeneren og TV-apparatet. Til dette er det vanlig å bruke en *set-top-boks*. En datamaskin kan også fungere i enkelte tilfeller, men IPTV-produkter lansert for et bredt marked bruker stort sett små bokser som kobles til TVen på samme måten som en videospiller. De set-top-boksene som blir brukt til IPTV tar imot digitale signaler og omformer

dette til noe som kan vises på TV-skjermen. Dette må ikke forveksles med analoge set-top-bokser som blir/ble brukt til å tolke analoge signaler fra kabel-TV-nettet.

Set-top-boksene er små datamaskiner i seg selv, og finnes i alle størrelser og fasonger, ofte med tilhørende fjernkontroll [STM04]. De små datamaskinene i set-top-boksene kjører ofte en enkel utgave av et operativsystem. Det vanligste er bokser som kjører Microsoft sin IPTV-løsning (Windows Media Center), eller bokser som kjører små utgaver av Linux.

2.3 SnapTV

Oppgaven vil bruke en IPTV-løsning fra SnapTV som grunnlag for en proof-of-consept implementasjon.

SnapTV sitt IPTV-system er basert på en *set-top-boks* til hvert enkelt TV-apparat som abonnerer på tjenesten. Denne kan da, via bredbånd, aksessere videostrømmer fra sentrale tjenere. Disse tjenerne lagrer sendinger fra TV-kanaler kontinuerlig og gir på denne måten video-on-demand mulighet til klientene. Videotjeneren kan også tilby video fra andre kilder enn TV-kanaler, som for eksempel DVD-filmer, Internett, tekst-TV, presentasjon av informasjon og reklame. Sluttbrukeren får på denne måten et svært fleksibelt og variert tilbud. Det er lagt opp til muligheter for både kjøp av fotballpakker, abonnement på kanaler og enkeltkjøp av spillefilmer. IPTV-systemene er tiltenkt både privat og offentlig bruk. For eksempel til borettslag eller hoteller. Så snart infrastrukturen er moden vil også SnapTV satse på å tilby IPTV over bredbånd.

2.4 Beskyttelse av IPTV

For å få lov av innholdsleverandører/opphavsrettsinnehaver til å sende TV-signaler over IP-protokollen, må tilgangen til innholdet beskyttes. Dette er blitt gjort tidligere i form av satellitt- og kabel-TV-systemer som også hadde *scrambling* av signalene. IPTV gir også utvidede muligheter som setter sterke krav til autentisering og autorisasjon. IPTV krever mye båndbredde, noe de fleste av dagens DSL-nett vil ha problemer med å håndtere. Derfor har det fram til nå vært mest brukt i lukkede nett, som i borettslag, hoteller eller båter. Båndbredden til vanlig DSL vil derimot øke og gi IPTV større behov for beskyttelse når hele verden i prinsippet kan ha tilgang til sendingene.

2.4.1 Eksisterende løsninger

IPTV er et fagområde i utvikling. Det dukker stadig opp nye metoder for videokomprimering, krav om høyere bildekvalitet, raskere oppdatering og flere

avanserte funksjoner. Det finnes mange aktører, spesielt internasjonalt. Det er gjort en del arbeid innenfor beskyttelse av multicast og gruppesikkerhet.

I 1998 opprettet The Internet Engineering Task Force (IETF) forskergruppen *Secure Multicast Group* (SMuG), som skulle ta for seg sikkerhet i IP-multicast. SMuG arbeidet svært tett med andre grupper innenfor samme område, som *Reliable Multicast Research Group* (RMRG), *Reliable Mulicast Transport* (RMT) og mer sikkerhetsrelaterte grupper som for eksempel *IP security protocol* (IPsec) [HD03]. Etter at SMuG ga ut en artikkel med *near standard* i 2000, delte gruppen seg opp i to deler: *Multicast security* (MSEC) og *the group security* (GSEC).

Internet Streaming Media Alliance (ISMA), har som mål å adoptere standarder innenfor streaming av lyd og bilde, utvikle disse og sette dem i drift. De har kommet med ikke-proprietære standarder for beskyttelse som for eksempel kan brukes i IPMP-X (se 2.5.3). Målet er en standard som gjør at innholdsdistributørene som følger denne kan kommunisere med avspillere, set-top-bokser og datamaskiner som har ISMA-støtte [All04].

Større leverandører av IPTV-løsninger har ofte sine egne proprietære løsninger. For eksempel har Microsoft fått med mange hardware-leverandører til å lage set-top-bokser som kun støtter deres teknologi. Andre hardwareprodusenter legger til rette for mer fleksible løsninger, men ofte er det begrenset med ressurser og muligheter.

2.5 Multimediastandarder

Dette avsnittet beskriver et sett med relevante multimediateknologier. Selv om vi ikke direkte benytter oss av disse teknologiene i oppgaven, danner dette korte overblikket en basis for senere diskusjon og forståelse.

2.5.1 MPEG

MPEG består av flere standarder for komprimering og distribusjon av digitale audio- og videosignaler. De viktigste audio- og videostandardene er MPEG-1, MPEG-2 og MPEG-4 (H.264). Mange av dagens distribusjonskanaler for digitalvideo bruker MPEG-2, for eksempel DVD-video, satellitt- og kabel-TV. De to sistnevnte benytter seg av MPEG-2 gjennom DVB-standarden (digital video broadcasting).

Prosessen med å komprimere (kode) og dekomprimere (dekode) en MPEG-strøm er asymmetrisk. Det vil si at komprimeringen er mer ressurskrevende fordi den har kompliserte algoritmer for å tilpasse seg datastrømmen for å kunne komprimere denne optimalt. Dekomprimeringen er derimot en relativt ukomplisert prosess som bare utfører et fast sett med operasjoner [Wat04]. Dette er en fordeling som fungerer bra fordi det er størst behov for dekodere som kan være implementert i billig forbrukerelektronikk. Dette har også ført

til at selve kodingen av MPEG-signalet ikke er standardisert, slik MPEG-datastrømmen og MPEG-dekodingen er. Koderen må bare implementeres slik at den produserer en datastrøm som kan dekodes av en standardisert avspiller.

MPEG-4 er en nyere og mer kompleks standard som blant annet kan komprimere data bedre enn MPEG-2 [Wat04]. Dette gjør MPEG-4 bedre egnet til områder hvor det er begrenset båndbredde, for eksempel til streaming over Internett, til mobile enheter eller IPTV. Ofte kan IPTV-systemer ha dedikerte lokale nettverk med høy båndbredde som gjør det mulig å distribuere video i MPEG-2-format, men selv her gir MPEG-4 høykvalitets video til samme båndbredde.

H.264, også kjent som AVC (advanced video encoding) eller MPEG-4 del 10, er en standard som konsentrerer seg om videokodingsaspektene ved MPEG-4. I motsetning til hele MPEG-4 standarden, som tar for seg mer enn bare videokoding, er H.264 sammenlignbar med MPEG-2, men med høyere ytelse og bedre funksjonalitet [Wat04].

2.5.2 MPEG datastrømmer

I tillegg til å beskrive datastrømmen og hvordan data dekodes, beskriver MPEG-standarden hvordan MPEG-data lagres og transporteres. Oppbygningen av en MPEG-datastrøm har det som kalles *elementary streams* (ES) i bunnen. En ES kan inneholde video-, audio- eller kontrolldata. For eksempel kan en enhet i ES-dataene bestå av et enkelt komprimert videobilde. ES-data prosesseres videre til en *packetised elementary stream* (PES) som består av datapakker av variabel lengde opp til 65536 bytes inkludert en 6 byte header [Fai]. PES-pakkene kan inneholde flere ES-pakker. Headeren i PES-pakkene identifiserer hva slags data den inneholder (audio, video, kontroll) og noe annen metainformasjon.

For å lagre eller transportere MPEG-dataene videre, må PES-pakkene multiplekses for å kunne overføres og avspilles samtidig. MPEG-2 definerer to metoder for multipleksing [Fai]:

- MPEG Program Stream (PS). Dette er multipleksede PES-pakker som egner seg for miljøer som er relativt feilfrie. For eksempel lagring, redigering og avspilling fra disk.

- MPEG Transport Stream (TS). TS består av PES-pakker som er delt inn i mindre TS-pakker med fast størrelse. En TS-strøm kan multiplekse flere datastrømmer for samtidig overføring av for eksempel separate TV-kanaler. TS brukes i miljøer hvor det kan oppstå pakketap og andre forstyrrelser. En TS-pakke har en fast størrelse på 188[2] bytes. Av

[2]Pakkestørrelsen på 188 i MPEG-TS ble valgt fordi man ønsket effektiv overføring av MPEG over ATM-nettverk. Cellene (ATM-pakker) i ATM-nettverk har en nyttelast på 47 bytes. En MPEG-TS-pakke kan derfor fordeles på fire ATM-pakker ($188 = 4 * 47$) [Fai]

dette er det en header på 4 bytes, hvor det viktigste parameteret er en *packet identifier* (PID), som beskriver hvilken PES-pakke denne TS-pakken er en del av, og gjør det mulig å demultiplekse TS-pakkene igjen for avspilling. Ved mottak må avspilleren synkronisere de forskjellige datastrømmene i transportstrømmen (for eksempel audio og video) fordi selve multipleksingen ikke er synkronisert i forhold til dette.

2.5.3 IPMP Extentions

IPMP (Intellectual Property Managment and Protection) er MPEG-standardens terminologi for DRM. MPEG-2 har hva man gjerne kaller *Conditional Access* (CA), som er tilpasset formålet ved distribusjon av videosignalet via satellitt eller kabel-TV. Da MPEG-4 først ble utarbeidet beskrev den IPMP i form av *hooks*. Dette originale MPEG-4 grensesnittet for IPMP-hooks spesifiserer bare hvilket system som er i bruk og hvor i datastrømmen et propritært system kan behandle data, ikke noe om selve IPMP-systemet. Hooks-systemet setter til side en IPI (Intellectual Property Identification) for å identifisere innhold på en proprietær måte (se tabell 2.1) og data-strømmer som kan fylles med informasjon om proprietære løsninger for å dekryptere innholdet. IPMP-modulene bygges opp som filtre som mediastrømmen kjøres igjennom [PW04].

En utvidelse av dette er IPMP-X (extentions) som inkluderer DRM-funksjonalitet og er mye mer fleksibelt. Her kan sluttbrukere benytte seg av beskyttet innhold på en transparent måte, som gjør at IPMP-systemet enkelt kan fornyes. IPMP-systemet er ikke standardisert i MPEG-4 fordi det fortsatt er mange slike systemer under utvikling, og det er lite sannsynlig at innholdsleverandørene vil velge et felles system [PE02]. IPMP-X er definert for MPEG-2 og MPEG-4.

I IPMP-X kan utvikleren av systemet definere et sett med *verktøy*. Disse tar seg av spesifikke operasjoner som autentisering, dekryptering eller vannmerking. Verktøyene registreres hos en sentral enhet med en unik ID. En MPEG-4-strøm vil da inneholde en IPMP-ES (elementary stream) som beskriver hvilke verktøy som er i bruk. IPMP-klienten vil benytte disse verktøyene, og eventuelt laste disse ned fra Internett hvis de ikke er tilgjengelig lokalt [PW04]. Slik modularisering av IPMP-funksjonene gjør systemet fleksibelt og man har mulighet til å bytte ut komponenter og oppdatere hvis man skulle finne svakheter, eller ny og bedre teknologi gjøres tilgjengelig [Gro].

2.5.4 MPEG-21

MPEG-21 definerer et stort rammeverk for levering og forbruk av multimedia. To viktige konsepter er defineringen av objektene i systemet (the Digital

Item) og brukerne som jobber med disse objektene [FK04]. Hensikten med rammeverket er å definere et felles grensesnitt for å muliggjøre kompatibilitet mellom ulike systemer og applikasjoner. Denne kompatibiliteten skal gjøre det mulig for brukere å utveksle, få tilgang til og manipulere digitale objekter på en transparent måte [BH].

MPEG-21 beskriver 12 deler som kan implementeres hver for seg. Noen av de viktigste delene, som er beskrevet i [BdWH+03], er definisjonen av selve rammeverket (1), deklarering og identifisering av objekter (2, 3), tilgangs- og rettighetskontroll i form av IPMP (4) og rettighetsspråk (5).

MPEG-21 er en ny omfattende standard, og det er et pågående arbeid både i standardiseringen og utvikling av rammeverk som støtter de forskjellige delene.

2.5.5 Real-time Transport Protocol

Real-time Transport Protocol (RTP) er en standard for robust overføring av sanntidsdata, som audio og video, for IP-nettverk. Protokollen beskriver tjenester for timing, deteksjon og korrigering av pakketap, identifisering av innhold og datakilde, og synkronisering. Opprinnelig var protokollen utviklet for multicast-konferanser, men har vist seg egnet til andre bruksområder som for eksempel IP-telefoni, webcasting og TV-distribusjon. RTP benytter seg av UDP-protokollen [Per03].

Secure RTP

Secure RTP (SRTP) er et rammeverk som kan sikre konfidensialitet og integritet for RTP-data og kontrollmeldinger, beskrevet i [BMN+04]. Den kan brukes til å sikre både unicast- og multicast-overføringer. Den definerer en header utenpå RTP-meldingene som inneholder nødvendige data for å beskrive de krypterte dataene. Disse header-feltene overføres ukryptert, noe som tillater header-komprimering.

AES i CTR-modus, som beskrevet i avsnitt 2.7, er standard konfidensialitetstransformasjon. Rammeverket kan også tilpasses andre kryptografiske transformasjoner. Hver melding blir individuelt kryptert slik at pakketap kan tolereres.

Integritetssjekk og meldingsautentisering er ikke pålagt av standarden, men det anbefales blant annet for å sikre mot *replay*-angrep [BMN+04] og tilføring av uønskede pakker i datastrømmen. HMAC-SHA1 er standard algoritme for meldingsautentiseringen. Dette sikrer både header og data mot endringer og forfalskning. Men samtidig øker det meldingsstørrelsen med 160 bits i tillegg til header-størrelsen som er på minst 128 bits.

2.6 Nettverkteknologi

IPTV er tett knyttet mot høy bruk av datanettverk. Vi vektlegger multicast, da de største utfordringene våre ligger her.

2.6.1 Multicast

I rapporten har vi brukt ordet multicast om IP-multicast, det vil si multicast over IP-protokollen. Multicast gjør det mulig å sende ut for eksempel videostrømmer til mange klienter uten at hver klient har en egen tilkobling til serveren [PD03]. For å begrense server- og nettverksbelastningen er dette en fornuftig måte å løse massedistribusjon av videostrømmer på. I praksis fungerer det slik at strømmen kopieres ved rutere der klientadressatene befinner seg på forskjellige destinasjoner. For at dette skal være mulig kreves det støtte i ruterene for multicast-rutingprotokoller. Klienter som er interessert i en multicast-gruppe gjør dette kjent for sine nærmeste rutere ved hjelp av Internet Group Managment Protocol (IGMP).

Multicast bruker UDP som transportprotokoll. Dette er som kjent en forbindelsesløs-protokoll og har ingen garantier for om pakkene blir levert og i hvilken rekkefølge klienten mottar dem.

For et lokalnettverk (innenfor en switch/hub) kan multicast-pakkene sammenlignes med broadcast. Til forskjell fra broadcast vil multicast kunne overføres over rutere hvis dette er konfigurert. Det finnes også switcher med *IGMP snooping*, som ikke uten videre sprer multicast-pakker til alle portene sine, men lytter etter IGMP-trafikk og sender multicast-trafikk bare til de klientene som har etterspurt dette.

En annen måte man kan begrense spredningen av multicast-pakker er ved bruk av TTL-feltet (time-to-live) for UDP-pakkene. En pakke blir ikke rutet videre med mindre TTL er større enn en bestemt terskelverdi. En liste med TTL-terskler som vist i tabell 2.2 er hentet fra [dG].

TTL	Scope
0	Begrenset til samme maskin. Spres ikke til nettverket. For eksempel til lokal testing av multicast applikasjoner.
1	Begrenset til samme subnet. Spres ikke over rutere.
< 32	Begrenset til samme organisasjon.
< 64	Begrenset til samme region.
< 128	Begrenset til samme kontinent.
< 255	Ubegrenset. Global.

Tabell 2.2: Multicast TTL-terskler

2.6.2 Any-source multicast

Any-source multicast (ASM) er betegnelsen på tradisjonell multicasting med en gruppeadresse i området 224/4 (IPv4) [AAMS01]. Det er ingen begrensning på hvem som kan sende data på en gruppeadresse. Slik kan man potensielt få konflikt ved at flere tjenester sender på samme adresse.

2.6.3 Source-specific multicast

For source-specific multicast (SSM) vil mottakeren legge til kildeadressen, i tillegg til gruppeadressen, når den melder seg inn i en multicast-gruppe. SSM er tildelt adresseområdet 232/8 (IPv4) [AAMS01].

Kombinasjonen av gruppe- og kildeadressen kalles en kanal (channel) [Bha03]. Multicast-mottakerne vil nå sende med kildeadressen i tillegg til gruppeadressen når de melder seg på en gruppe. Multicast-ruterene vil da vite hvor multicast-kilden er. Men kilden trenger fortsatt ikke vite noe om mottakerene. Dette vil også gjøre det vanskeligere å forfalske eller komme i konflikt med eksisterende multicast-sendinger fordi man spesifikt ber om data fra en bestemt avsender.

SSM løser også problemet med at adresseområdet på multicast IPv4 ikke er så stort, slik at det kan bli konflikter ved WAN multicasting [WRSR04].

De fleste nyere operativsystem har støtte for SSM. Systemet trenger støtte for IGMPv3[3] for SSM over IPv4 og MLDv2[4] for IPv6.

2.6.4 Beskyttelse av multicast

Når multicast-data skal beskyttes er det i hovedsak to problemer som må løses:

1. Krypteringsmetoden må tolerere pakketap (*packet loss*).

2. Krypteringsnøklene må kunne skiftes sømløst i strømmen og må distribueres til aktuelle klienter.

Kryptering av multicast-strømmer

En multicast-strøm vil være lossy fordi den baserer seg på UDP, og det må kunne aksepteres pakketap. Hver enkelt multicast-pakke må derfor være kryptert som en selvstendig melding uten at pakken trenger å være avhengig av foregående pakker, som potensielt kan forsvinne, for å dekrypteres.

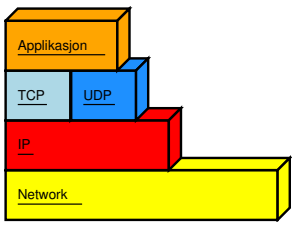
[3] Internet Group Management Protocol, versjon 3, RFC3376
[4] Multicast Listener Discovery, versjon 2

Nøkkeldistribusjon og nøkkelbytter

Siden multicast distribuerer de samme krypterte datapakkene ut til hver klient, trenger hver enkelt klient den samme nøkkelen (kalt *gruppenøkkel*) for dekryptering. Distribusjon av disse nøklene må skje over en sikker kanal. Bytting av nøkkler bør være mulig for å kunne hindre uautorisert tilgang fra for eksempel tidligere abonnenter. Nøkkelbytte vil også til en viss grad vanskeliggjøre kryptoanalytiske angrep på strømmen, og kan til en viss grad kompansere for svake krypteringsalgoritmer. Hvis en av klientene skal sperres for tilgang til dataene fra multicast-gruppen må nøkklene hos alle klientene byttes ut. Ved (svært) mange klienter kan dette potensielt bli en langsom og ressurskrevende operasjon.

Andre aspekter

Beskyttelse av multicast kan gjøres på forskjellige lag i OSI-modellen. Teknologiene som er nevnt i avsnitt 2.4.1 er basert på standardisering av beskyttelse basert på IP- og TCP/UDP-lagene. Dette løser flere svakheter framfor å legge beskyttelsen i applikasjonslaget. En høynivå-kryptering vil ikke sikre overføringen mot analyse av trafikken og transportinformasjonen i pakkene. Beskyttelse på høyere nivå gir heller ingen mulighet for å avgrense multicast-klientene med tanke på hvor langt vekk fra kilden det gis tilgang til strømmen. Derimot har beskyttelse i applikasjonslaget den fordelen at det kun kreves generell multicaststøtte i ruterene og nettverkstopologien. På figuren 2.5 ser vi *applikasjonslaget* over eventuelle UDP og TCP forbindelser i TCP/IP modellen sin lagmodell.



Figur 2.5: TCP/IP modell

2.7 Kryptografi

Dette avsnittet vil ta for seg deler av kryptografiteorien som vi senere vil bruke til å velge og diskutere metoder som passer til vårt problemområde. Dette er ikke en oppgave om kryptografi, men kryptografi er et meget viktig element i flere av problemstillingene. Vår hensikt er her å gjennomgå noen viktige elementer innen relevante områder. Dette gjelder spesielt en gjennomgang av symmetriske chifferalgoritmer som brukes i krypteringen

av videostrømmer, og i tillegg asymmetriske krypteringsmetoder til bruk i autentisering av klientsystemene.

2.7.1 Chifferalgoritmer

Blokkchiffere er de elementære byggeblokkene for symmetrisk kryptering hvor avsender og mottaker deler en felles nøkkel. Et blokkchiffer er definert med en fast blokkstørrelse og nøkkellengde. I dagens chifferalgoritmer, som for eksempel AES, kan dette være 128 bits (16 byte) for begge. Nøkkellengden kan økes utover dette uten å endre blokkstørrelsen. Med den samme nøkkelen, og lik klartekstblokk, vil et blokkchiffer kryptere denne blokken til et identisk kryptogram hver gang.

Det finnes også noe kalt strømchiffer (stream cipher). Dette krypterer en bit, byte eller word av gangen og vil kryptere denne enheten til forskjellig verdi for hver kryptering. Det finnes flere strømchiffere, men få av dem er implementert i tilgjengelige kryptobiblioteker. Gcrypt, som vi beskriver i avsnitt 5.2.3, implementerer kun ARCFOUR (RC4) chifferet. I følge [Sch96] er strømchiffere best egnet i hardware-implementasjoner fordi det der er mer naturlig å kryptere en og en bit når de går gjennom systemet. Man må også være forsiktig når man implementerer strømchiffere fordi det er lett å gjøre systemet usikkert. Man er avhengig av en pålitelig kilde for tilfeldige tall (random generator).

Det er flere aktuelle blokkchiffere og vi vil kort beskrive noen av dem her:

AES

AES (Advanced Encryption Standard) adopterte chifferalgoritmen Rijndael etter en etterlysning av forslag til en ny kryptografistandard for den Amerikanske regjeringen. Kryptografieksperten Bruce Schneier beskriver AES som det *sikre* valget du i hvert fall ikke får sparken for om du velger. [FS03] Algoritmen ble valgt som standard blant annet fordi den er veldig rask både i software og hardware implementasjoner. AES er definert med en nøkkellengde på 128, 192 eller 256 bits. 128 er standard nøkkellengde og blokkstørrelsen er 128 bits.

Twofish

Twofish er utviklet av et team bestående av blant annet Schneier, og var et av forslagene til AES-standarden. Den ble slått av Rijndael i hastighet ved en nøkkellengde på 128-bits, men med lengre nøkler er den raskere enn Rijndael [FS03] (i teorien, se C.3). Standard nøkkellengde er 256 bits.

CAST5

CAST5 (også kalt CAST-128) var en AES-utfordrer utviklet i Canada. Den brukes blant annet som standardchiffer i programmet GnuPG[5].

Serpent

Dette går for å være den sikreste av AES-konkurrentene, men den er også den tregeste.

DES

DES er et chiffer utviklet av IBM tidlig på 70-tallet. Den har vært amerikansk krypteringsstandard siden 1976, og begynner nå og dra på årene. Den største svakheten er nøkkellengden på 64-bits, av disse er det bare 56-bits av nøkkelen som er i effektiv bruk, 8 av bitene brukes (eller kan brukes) som paritetsbits. Sammenlignet med nyere algoritmer, som AES, er den også treg, i hvert fall i software-implementasjoner. I hardware-implementasjoner klarer den seg bedre [FS03]. Og på grunn av dens fortsatt relativt utstrakte bruk, blant annet i noen IPTV set-top-bokser, kan den ikke avskrives helt enda.

Chifferet kan også brukes i en form som kalles 3DES, det vil si at DES-chiffere brukes tre ganger etter hverandre med en nøkkellengde på 168-bits (3x56). Dette gjør data kryptert med DES sikrere, men medfører at operasjonen bruker tre ganger så lang tid.

XTEA

TEA (Tiny Encryption Algorithm) [WN94] eller XTEA (en forbedring av TEA) [WN97] er et enkelt blokkchiffer som kan implementeres på bare noen få linjer. Etter anbefaling så vi nærmere på dette chifferet for å se om dette kunne være en enkel og lite krevende algoritme som kan brukes til å kryptere store menger data som multimedia består av. Se avsnitt C.3 for en nærmere beskrivelse av chifferet.

2.7.2 Blokkchiffermodus

Et blokkchiffer kan ikke kryptere mer data enn blokkstørrelsen. Hvis dataene man vil kryptere er større enn dette, og det er de gjerne, så må man kjøre blokkchifferet i et av flere modus.

Hensikten med kryptering er å sikre konfidensialitet, integritet og autentisitet. Blokkchiffere sikrer kun konfidensialiteten til dataene. Blokkchiffermodus endrer ikke dette [Sch96], så hvis man vil sørge for integritet og

[5] GnuPG, Open Source alternativet til Pretty Good Privacy (PGP) http://www.gnupg.org/

autentisitet for dataene man krypterer må man kombinere et blokkchiffermodus med andre mekanismer, som for eksempel *Message Authentication Codes* (MAC), som lager en signatur av dataene ved hjelp av en nøkkel. Uten dette vil en angriper kunne endre, slette eller legge til pakker selv om man ikke kjenner nøkkelen dataene ble kryptert med.

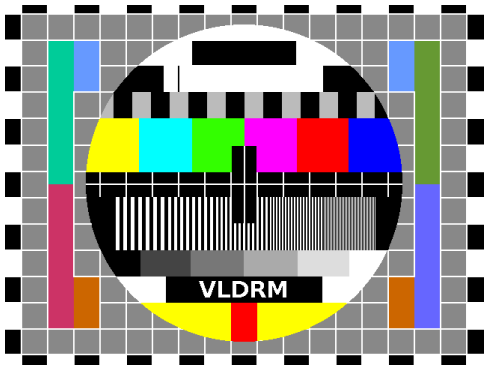
For kryptering av multimediadata vil vi skille mellom for eksempel et videokonferansesystem som vil ha behov for høy konfidensialitet, integritet og autentisitet, og videodata i et IPTV-system som vi beskriver, som ikke har det samme behovet.

ECB

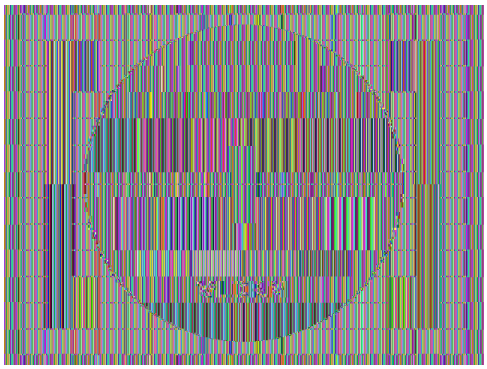
ECB (Electronic Codebook) er det enkleste chiffermodus. Her brukes chifferfunksjonen direkte på hver blokk. Dette medfører at ved bruk av samme nøkkel så vil to identiske blokker krypteres til den samme chifferteksten [Dwo01].

Fordelen med ECB er at rekkefølgen på krypteringen ikke spiller noen rolle og man kan kryptere forskjellige blokker parallelt. En databasefil kan krypteres i ECB og aksesseres tilfeldig. Men dette gjør det samtidig lettere for en uautorisert tredjepart å endre, slette eller legge til data. Man er også sårbar for *replay attacks*, da en tredjepart som lytter til en kommunikasjon kan gjenta dette ved en senere anledning. [FS03] anbefaler at man holder seg langt unna ECB for alt annet enn kryptering av enkeltpakker mindre enn blokkstørrelsen på chifferet.

I tekst eller videodata vil dataene ofte være strukturert og ha mange like blokker. ECB klarer da ikke å skjule mønster i lange meldinger. Hvis meldingen ikke er lenger enn blokkstørrelsen er ikke dette noe problem. ECB-modusen sin manglende evne til å skjule mønstre i de krypterte dataene kan illustreres godt med følgende eksempel. Bilde i figur 2.6 kan sees kryptert med AES i ECB-modus i figur 2.7. Her kan man tydelig se pixel-mønsteret i det krypterte bilde.



Figur 2.6: Ukryptert originalbilde



Figur 2.7: Bilde kryptert med AES i ECB-modus

Chiffermoduset trenger padding opp til hele blokkstørrelser.

CBC

CBC (Cipher Block Chaining) løser problemet med ECB ved å benytte seg av resultatet fra forrige krypterte blokk i krypteringsprosessen. Det vil si at klartekstblokken blir XOR-et sammen med foregående kryptogramblokk før den krypteres med chifferet. Ligning 2.1 og 2.2 viser henholdsvis kryptering og dekryptering, hvor C_i er gjeldene kryptogram, C_{i-1} er forrige kryptogram, P er klarteksten og E_k er krypteringsfunksjonen med nøkkelen k [Dwo01].

$$C_i = E_k(P_i \oplus C_{i-1}) \tag{2.1}$$

$$P_i = C_{i-1} \oplus D_k(C_i)) \tag{2.2}$$

For den første blokken som krypteres må man generere en verdi for C_{i-1}, kalt initialiseringsvektor (IV). Dette beskrives nærmere nedenfor. Ved hjelp av dette vil mønster i klarteksten skjules av lenkingen, men lange meldinger kan fortsatt ha mønster. I et problem kalt "the birthday paradox" [Sch96] vil det kunne oppstå identiske blokker etter et visst antall blokker har blitt kryptert med samme IV og nøkkel. Generelt kan dette skje etter $2^{(m/2)}$ blokker er kryptert (m = blokkstørrelse). Med 64-bit blokkstørrelse er dette ca. 34 gigabytes, med 128-bit blokkstørrelse blir dette tallet *veldig* stort og vi har ikke tenkt til å gjengi det her. Så man kan kryptere svært mye data før dette problemet oppstår.

CBC er et *selvsynkroniserende* chiffermodus. En 1-bit feil i kryptogrammet fører til at klartekstblokken blir uleselig og en 1-bit feil overføres til neste klartekst blokk. Deretter fortsetter den uten feil [Sch96].

[FS03] anbefaler CBC som spesielt egnet for kryptering av statiske filer og generelt for softwarebasert kryptering.

Som en oppfølging av illustrasjonen i figur 2.7 viser figur 2.8 bilde fra figur 2.6 kryptert i CBC-modus. Her er mønsteret skjult og det ser ut som bilde består av tilfeldige pixler.

Figur 2.8: Bilde kryptert med AES i CBC-modus

Chiffermoduset trenger padding opp til hele blokkstørrelser.

CFB

Med CFB (Cipher Feedback Mode) vil chifferet fungere som et strømchiffer, og data kan krypteres i mindre størrelser enn blokkstørrelsen. Krypteringen kan starte med en gang man får data tilgjengelig uten å fylle en hel blokk [Dwo01]. Dette kan man utnytte hvis man for eksempel har en Telnet-applikasjon hvor ett og ett tegn skal overføres av gangen.

Som i CBC må man bruke en IV. I CFB-modus bruker man en nonce (number used once) som må være unik og endres for hver melding.

Når bit-feil oppstår i CFB-modus vil omfanget av feilen være avhengig av den interne registerstørrelsen (segmentstørrelse), det vil si den enheten som blir kryptert av gangen. I 8-bits CFB som kan brukes for å kryptere ASCII-tegn vil en 1-bit-feil føre til feil i de neste 9 bytes i klarteksten [Sch96]. Generelt vil en feil i n-bits CFB-modus føre til feil i de neste $m/n - 1$ blokkene, der m er blokkstørrelsen.

En fordel med CFB er at man bare trenger å implementere krypteringsfunksjonen, ikke dekrypteringsfunksjonen (se ligning 2.3 og 2.4).

$$C_i = P_i \oplus E_k(C_{i-1}) \tag{2.3}$$

$$P_i = C_i \oplus E_k(C_{i-1}) \tag{2.4}$$

CFB trenger i utgangspunktet ingen padding. Størrelsen på dataene som skal krypteres må gå opp i segmentstørrelsen. CFB er i software som regel implementert med en segmentstørrelse på en byte, og det er som oftest den minste enheten man krypterer.

OFB

OFB (Output Feedback Mode) har mange likhetstrekk med CFB. Den trenger kun krypteringsfunksjonen. I OFB må man være forsiktig med å ikke bruke samme IV med samme nøkkel [FS03]. Da dette chiffermoduset er lite støttet i krypteringsbiblioteket vi beskriver i avsnitt 5.2.3 ser vi ikke noe nærmere på dette her.

CTR

CTR (Counter Mode) er Schneier sin favorittmodus [FS03]. Denne er også veldig lik CFB og OFB. Man genererer her en nonce som man setter sammen med en teller. I stede for å bruke tidligere kryptogrammer som input til algoritmen brukes denne telleren. [FS03] anbefaler denne hvis man må velge mellom OFB/CFB og CTR. Også her må man være forsiktig med å ikke bruke samme *nonce* flere ganger.

Man trenger bare å implementere krypteringsfunksjonen, ikke dekryptering. Det genereres en IV som settes sammen med en teller. Lengden på disse dataene er lik blokkstørrelsen til chiffere. Denne blokken blir så kryptert med gjeldene nøkkel og XOR-et med klarteksten for å lage kryptogrammet, se ligning 2.5. Dekryptering gjennomgår samme prosess og XORes med kryptogrammet for å få tilbake klarteksten, se ligning 2.6. Se avsnitt 6.2 for en mer detaljert beskrivelse av hvordan vi implementerer CTR-modus i praksis.

$$C_i = E_k(nonce|teller) \oplus P_i \tag{2.5}$$

$$P_i = E_k(nonce|teller) \oplus C_i \tag{2.6}$$

I og med at CTR gjør chiffer til et strømchiffer trenger man ingen padding.

Padding

Blokkchiffermodus som ECB og CBC kan bare kryptere data med lik størrelse som blokkstørrelsen. Det er derfor nødvendig å fylle opp meldinger som krypteres opp til blokkstørrelsen. Denne paddingen må også være reversibel. En måte å gjøre dette på er beskrevet i [Hou99]. Her paddes de siste $m-(l\%m)$ bytene med verdien $m-(l\%m)$, hvor m er blokkstørrelsen og l er lengden på dataene (som er mindre enn blokkstørrelsen). Krypterte data som blir behandlet på denne måten vil få en størrelse som er minst en blokkstørrelse større enn de ukrypterte dataene selv om dataene går opp i blokkstørrelsen. Dette er fordi man skal kunne ha mulighet til å fjerne paddingen igjen.

Hvis man ikke kan øke størrelsen på kryptogrammet i forhold til klarteksten ved å bruke padding, kan den siste blokken krypteres annerledes.

Initialiseringsvektor

De fleste chiffermodi, unntatt ECB, trenger en form for initialiseringsvektor (IV). Dette er egentlig ikke noe annet en *dummy*-kryptogramblokk som brukes for å randomisere det første kryptogrammet slik at ikke identiske klartekstblokker krypteres til identiske kryptogram. I CBC brukes IV som første C_{n-1} (se ligning 2.1) og CFB/OFB brukes den til å fylle de interne feedback-registrene. IV-en trenger ikke å være hemmelig. Den kan for eksempel overføres eller lagres som den første blokken i de krypterte dataene. Det er kritisk at IV-en er unik for hver nøkkel. Hvis ikke kan man risikere å lekke informasjon enten om klarteksten eller nøkkelen.

IV-en kan genereres på forskjellige måter [FS03]:

- Fast verdi. Dette er et dårlig valg. Den første blokken i dataene vil da ha samme svakhet som data kryptert i ECB-modus. Hvis dataene i tillegg for eksempel begynner med en kjent verdi (som mange kommunikasjonsprotokoller ofte gjør) vil det være lettere for en tredjepart å finne krypteringsnøkkelen.

- Teller. Også her vil den første pakken være utsatt hvis telleren begynner på samme verdi hver gang.

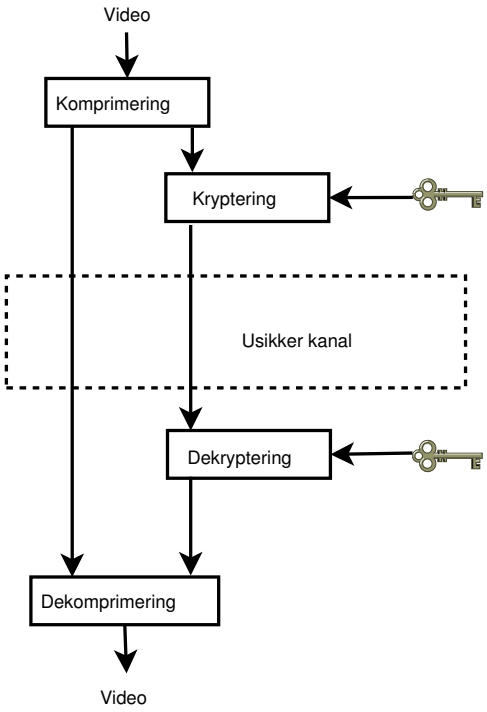
- Tilfeldig. Denne verdien kan genereres av en pseudo-slumptallsgenerator, være en *timestamp* eller en annen tilfeldig verdi.

- Nonce. (Number used once) Et nonce er en spesiell form for en tilfeldig verdi. Det er veldig viktig at dette tallet ikke brukes om igjen med samme nøkkel. Et nonce genereres og settes til tellerverdien ved CTR og IV for CFB.

2.7.3 Selektiv kryptering

Vanlige krypteringsmetoder egner seg mindre til å kryptere multimediadata, da de fleste multimediaoverføringer har mye data, liten verdi og har strenge sanntidskrav [SB98]. Ved *naiv kryptering* [FK04], vil det si at man krypterer (med for eksempel AES) alle data uavhengig av formatet. Som nevnt tidligere må man se på behovet og kostnaden ved beskyttelse i forhold til verdien av det som beskyttes. En videokonferanse vil for eksempel ha større behov for beskyttelse enn en spillefilm.

Siden videostrømmer blir *ubrukbare* i forhold til underholdningsverdi når deler av bilde forsvinner, kan en god løsning være å kryptere bare deler av videostrømmen. Slik kan man oppnå høyere effektivitet og bedre skalerbarhet enn ved kryptering av hele strømmen. Se figur 2.9 for hvordan dette i grove trekk løses i forbindelse med komprimeringsprosessen. Det finnes flere former for selektiv kryptering [LSKG03]:



Figur 2.9: Selektiv kryptering. Bare deler av strømmen krypteres.

- Headerkryptering. Det har blitt gjort forsøk på kryptering av kun *header*, men headere er ofte enkle å rekonstruere. Dette alene er ingen god løsning.

- Forutseende basert selektiv kryptering. Kryptering av I-frames i en MPEG strøm kan fungere bra, det er fremdeles mulig å kryptoanalytisk finne nok informasjon til å danne seg ett bilde av innholdet. Derimot er en stor del av strømmen I-frames (25-50%) [LSKG03]. Dette gjør det mindre hensiktsmessig fremfor andre metoder som kan kryptere en mindre del av strømmen.

- DCT-koeffisient. [SB98] beskriver algoritmen VEA (Video Encryption Algorithm) som bruker en enkel XOR mellom *sign bits av DCT* (descrete cosinus transformasjon) og en hemmelig nøkkel. Forsøkene til [SB98] har funnet ut at DCT-datadelene av en MPEG-strøm er ca. 13,8% av strømmen. Dette gir en god skaleringsmulighet fremfor å kryptere hele strømmen.

- Zig-Zag scan, omrokering. Denne metoden stokker om rekkefølgen DCT koeffisientene blir skannet. Dette gir en overhead på komprimering, og løsningen er derfor ikke gunstig.

- One Time Pad. En kryptert del av MPEG-filmen brukes til å forandre resten av den ukrypterte MPEG-strømmen ved XOR mellom disse to. Slik at den krypterte delen fungerer som kryptering (pad) også for den ukrypterte delen.
- Tilfeldig kryptering (Random Corruption) Metoden brukes for å kryptere en tilfeldig del av MPEG-strømmen, uavhengig av innhold og format på strømmen. Dette gir større muligheter for å kryptoanalytisk kunne knekke krypteringen.
- Skalerbare systemer. I [GMDS98] brukes det forhåndslagret kryptert video som er lett tilgjengelig for bruker. Systemet her legger vekt på caching av video nærmere brukeren enn på serveren. Videoen trenger derimot en nøkkel for å kunne dekrypteres realtime.

Noe som både kan være en ulempe og en fordel med flere av disse metodene er at de er sterkt knyttet til komprimeringsfasen. [Tan96] beskriver en hvordan krypteringen blir en del av komprimeringsfasen, slik at komprimering og kryptering (samt dekomprimering og dekryptering) blir en og samme prosess. Dette øker klart effektiviteten. I de tilfeller hvor datastrømmen allerede er komprimert er man nødt til å ty til andre metoder for kryptering av dataene. I forhold til kryptering er komprimeringen (kodingen) av multimediadatane en mer krevende prosess [FK04].

2.7.4 Sertifikater

Digitale sertifikater har forskjellige bruksområder, på WWW oftest brukt til verifisering av en server. Derimot fungerer det også utmerket å verifisere klienter ved hjelp av sertifikater.

Klientautentisering ved hjelp av et sertifikat kan foregår slik [Sch04]:

- Klienten sender sertifikatet sitt til serveren.
- Serveren kontrollerer sertifikatets gyldighet.
- Serveren kontrollerer om den stoler på CA (Certificate Authority) som har signert sertifikatet.
- Serveren dekrypterer signaturen med CA sin offentlige nøkkel for å se hvem sertifikateieren er og få denne sin offentlige nøkkel.
- Serveren sender en *challenge* til klienten for at serveren kan vite med sikkerhet at klienten er den som den gir seg ut for.
- Klienten svarer på *challenge* og serveren vet da at klienten er den den utgir seg for, siden den må være i besittelse av den private nøkkelen for å kunne svare.

Det finnes forskjellige variasjoner av denne autentiseringen, i henhold til hva en trenger å bekrefte/sikre i overføringen.

X.509

X.509 versjon 3 er det mest brukte sertifikatformatet på Internett i dag [Cro02]. Det er dette som blir brukt som format for sertifikater i nettlesere SSL/TLS og til etablering av *trust* i PKI (Public Key Infrastructure) hierarkier. X.509 er basert på X.500 som var en telekommunikasjonsrettet katalogtjeneste [Wike]. X.509 er definert som et rammeverk for identifisering innenfor denne katalogtjenesten. Hvert X.509 sertifikat inneholder innehaverens offentlige nøkkel. Sertifikatet er signert med den private nøkkelen til en CA. X.509-spesifikasjonen setter ingen krav til valg av krypteringsalgoritme, men anbefaler bruk av RSA.

Innholdet i et X.509 sertifikat [Cro02]:

- Sertifikat
  - Versjon (1,2 eller 3)
  - Serienummer, heltall unikt hos utstedende CA.
  - Identifikasjon av signaturalgoritme, også gjentatt til slutt i sertifikatet.
  - Utsteders navn, X.500-navnet til utstedene CA.
  - Gyldig tidsrom
    - Ikke før
    - Ikke etter
  - Navnet på den/det sertifikatet er utstedt til (sertifikateier).
  - Sertifikateiers offentlige nøkkel
    - Offentlig nøkkel algoritme
    - Sertifikatinnehavers offentlige nøkkel
  - Utsteders unike identifikator (versjon 2-3)
  - Emne unik identifikator (versjon 2-3)
  - Utvidelser (versjon 3)
- Identifikasjon av signaturalgoritme.
- Sertifikatsignatur, hash av hele sertifikatet, kryptert med signerende CA sin private nøkkel.

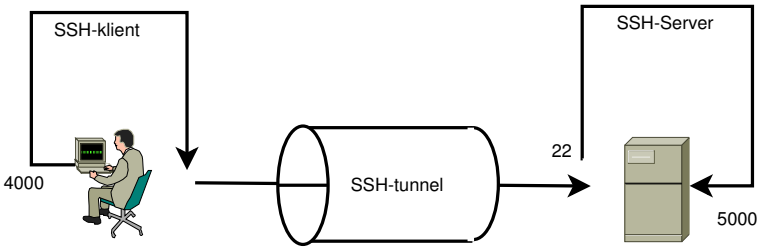
Fordi sertifikatet er signert er det ikke nødvendig å beskytte det på annen måte enn at signaturen bør kontrolleres før bruk. Alle brukere av sertifikatet kan kontrollere sertifikatet ved hjelp av CA sin offentlige nøkkel. Ingen andre enn utstedene CA kan endre sertifikatet uten at det vil bli oppdaget. [Sta03]

Hvordan nøklene og sertifikatene lages, distribueres og kontrolleres blir ofte referert til som PKI i større sammenhenger. Det blir laget et *trust* hierarki, med en CA på toppen som signerer andre sertifikater med sin private nøkkel, fordi disse respektive sertifikatholderene er til å stole på. Disse signerte sertifikatene kan igjen signere andre sertifikater osv. Dette gjør at hvis en klient stoler på det øverste sertifikatet i hierarkiet, kan den også stole på alle de underliggende. Ved hjelp av nøkkelsettet kan data krypteres med den offentlige nøkkelen, og kun åpnes med den private nøkkelen

2.7.5 SSH-tunnel

En SSH-tunnel er en meget enkel måte å bruke ukrypterte tjenester over en kryptert kanal. Når en slik tunnel blir satt opp lager du en sikker kanal mellom en lokal maskin og en fjern maskin (kun mellom to maskiner). Etter at den sikre tunnelen er satt opp kan du ved å spesifisere portene til tunnelen konfigurere andre tjenester til å kommunisere igjennom tunnelen. Dette systemet blir ofte brukt til Telnet, SMTP, FTP, VNC og andre ukrypterte nett-tjenester. I figur 2.10 er det satt opp en SSH-tunnel fra port 5000 på serveren til port 4000 hos klienten. Klienten kan da koble seg opp mot port 4000 på sin lokale maskin og trafikken blir tunnelert frem til port 5000 på serveren.

SSH-protokollen består av tre komponenter: Transportlagsprotokollen autentiserer serveren og sørger for konfidensialitet og integritet på overføringen. Brukerautentiseringsprotokollen autentiserer klienten overfor serveren. Forbindelsesprotokollen multiplekser den krypterte tunnelen til flere logiske kanaler [ID].



Figur 2.10: SSH-tunnel fra port 5000 til 4000.

SSH kan brukes til å simulere VoD (unicast) se avsnitt C.1.

Del II

Systembeskrivelse

Kapittel 3

Kravspesifikasjon

Kravspesifikasjonen er basert på hva SnapTV så som første steg i beskyttelse av sin IPTV løsning. Dette er i overensstemmelse med utviklingsveikartet til SnapTV [Bøh04], og problemstillingen i oppgaven vår.

3.1 Overordnet systembeskrivelse

Det skal lages et system for beskyttelse av innholdet i en IPTV-løsning. Dette blir laget for å tilfredsstille rettighetsholdere av media-innholdet som skal distribueres.

Viktige egenskaper:

- Beskyttelse av datastrømmen fra video-server til klient.
- Beskyttelse av multicast-strømmer.
- Transparens i forhold til avspiller og mediaserver.
- Rammeverk for kontrollering av tilgang til innhold.
- Distribusjon av krypteringsnøkler.
- Det skal benyttes kjente løsninger og *standarder* på de områdene dette finnes. For eksempel sertifikater, lisensformat, kryptering og programvarebibliotek.

3.2 Rammekrav og forutsetninger

3.2.1 Ytelse

1. Systemet skal legge til så lite prosesseringsoverhead som mulig.

2. Tidsforsinkelse ved bytting av multicast- og unicast-strømmer skal ikke overstige 1 sekund.

3. Løsningen er beregnet for lokalnettverk. Internettlinjer med lav båndbredde vil ikke bli tatt hensyn til.

3.2.2 Sikkerhet

1. Overføring av kontrolldata bruker en sikker kanal.

2. Alle mediastrømmer skal ha krypteringsmulighet for å sikre overføring av innhold mot uautorisert tilgang.

3. Sikring av klientenheten mot fysisk innbrudd er utenfor denne oppgaven.

3.2.3 Transparent mediahåndtering

1. Grensesnittet mot videokildene skal kunne håndtere forskjellige typer videokilder.

2. Grensesnittet mot videokildene skal være konstruert på en slik måte at videokildene kan byttes ut. F.eks. nye video codecs.

3.2.4 Pakketap

1. Ved multicast skal strømmen kunne tolerere pakketap, dette skal kun i verste fall gi hakk i bilde. Krypteringsalgoritmene må ta hensyn til dette.

2. Unicast-strømmer skal ikke tillate pakketap. Bufferhåndtering overlates til avspilleren.

3.2.5 Initielle data

1. Server
   (a) Nøkkelpar og signert sertifikat
   (b) Klientinformasjon
      i. Offentlig nøkkel til alle klientene.
      ii. Bruker-ID (navn etc.)
      iii. Tilgangsinformasjon (Informasjon om hva hver enkelt klient har lov til å se på)
      iv. Betalingsmåte; pay-per-view, forhåndsbetalt eller andre betalingsmåter. Dette kan plusses på/trekkes fra i database eller brukersystem etterhvert.

(c) Adresse og informasjon om webservere og multicast-servere.

2. Klient

   (a) Nøkkelpar

   (b) Serverinformasjon

      i. Sertifikat, signert av serveren. Brukes av serveren for å autentisere klienten.

      ii. Serverens offentlige nøkkel. Brukes av klienten til å autentisere serveren.

   (c) Adresse (IP og port) til serveren (servermodulen)

3.3 Funksjonelle egenskaper

3.3.1 Overføringen

1. Overføringen skal være multicast eller unicast.

   (a) Unicast-overføringen må krypteres, innholdet tilbys fra webserver.

   (b) Multicast-overføringen må krypteres, innholdet tilbys fra streamingserver.

2. Kryptering

   (a) Brukerautentisering må være kryptert.

   (b) Nøkkel-/lisensutveksling må være kryptert (så ingen kan stjele nøklene eller utgi seg for å være en av partene).

   (c) Krypteringen må være transparent, slik at avspiller/server kan byttes ut uavhengig av beskyttelsen.

   (d) Krypteringen av multicast-strømmer bruker samme nøkkel for alle mottakere. Denne nøkkelen må byttes ved gitte intervaller.

   (e) Kryptering av mediastrømmer er valgfri. Kanaler som ikke krypteres vil enten kjøres utenom hele systemet eller kunne kjøres ukryptert gjennom systemet.

3.3.2 Klientmodul

1. Sesjonshåntering

   (a) Opprette en sikker kanal mot servermodulen.

   (b) Sende autentiseringsinformasjon til servermodul.

2. Dekryptering

   (a) Be serveren om krypteringsnøkkel for gjeldende datastrøm.
   (b) Håndtere nøkkelbytte initiert av server.
   (c) Dekryptere data med tilgjengelig nøkkel.
   (d) Klienten må ha et lite nøkkellager (i minne), slik at den kan utnytte de sist brukte korttidslisensene så lenge de er gyldige. Dette vil gi rask *zapping*.

3. Multicast-klient
   (a) Mottar multicast-strøm.

4. HTTP-proxy
   (a) Skal fungere som HTTP-proxy for mediaavspilleren.
   (b) Mottar HTTP-forespørsler fra mediaavspilleren.
   (c) Hvis unicast: videresender forespørsel til servermodul
   (d) Videresender dekrypterte data tilbake til mediaavspilleren.

3.3.3 Servermodul

1. Sesjonshåndtering
   (a) Oppretter sikker forbindelse med klientmodulen på forespørsel fra denne.
   (b) Autentisere klienter

2. Lisenshåndtering
   (a) Kontrollerer om klienten har tilgang til forespurt innhold.
   (b) Genererer innholdslisenser

3. Kryptering
   (a) Krypterer innhold før videresending.
   (b) Håndtere og distribuere krypteringsnøkler.

4. Multicast-server
   (a) Multicast-strømmer må spesifiseres i en konfigurasjonsfil.
   (b) Multicast-strømmene er i utgangspunktet sanntidsstrømmer og vil motta kildesignalet gjennom en UDP-strøm fra en mediaserver. Disse UDP-strømmene videresendes som multicast-strømmer.

5. HTTP-proxy
   (a) Unicast strømmer initialiseres fra avspiller hos klient.
   (b) Forespørslene videresendes til en webserver.
   (c) Data mottatt fra webserveren videresendes til kryptering.

3.4 Krav til dokumentasjon

- Teknisk dokumentasjon av systemet.
- Dokumentert kildekode/API (Doxygen). Kildekoden dokumenteres ved å inkludere formaterte kommentarer som programmet Doxygen kan bruke til å generere API-dokumentasjon. Syntaks for Doxygen er Java-Doc-basert.
- Enkel installasjons- og brukerveiledning for klient/server. Beskriver hvordan man fra kildekoden kompilerer og konfigurerer systemet slik at det kan fungere som tiltenkt.

Kapittel 4

Design

Denne delen av rapporten beskriver overordnet design av prototypen. Prototypen har fire formål:

- Hjelpe oss med å besvare flere av del-problemstillingene i oppgaven.
- Tilfredsstille behovet for et beskyttelsessystem for eksisterende IPTV-system.
- Undersøke systemets funksjonalitet og ytelseskrav.
- Belyse uforutsette problemstillinger.

4.1 Arkitektur

Arkitekturen i systemet er basert på en klient/server-løsning. Det er fire viktige overordnede komponenter i vår foreslåtte arkitektur. Hvor to av dem er fokus for våre problemstillinger, og de to endepunktene er komponenter som allerede er på plass i systemet. Komponentene er:

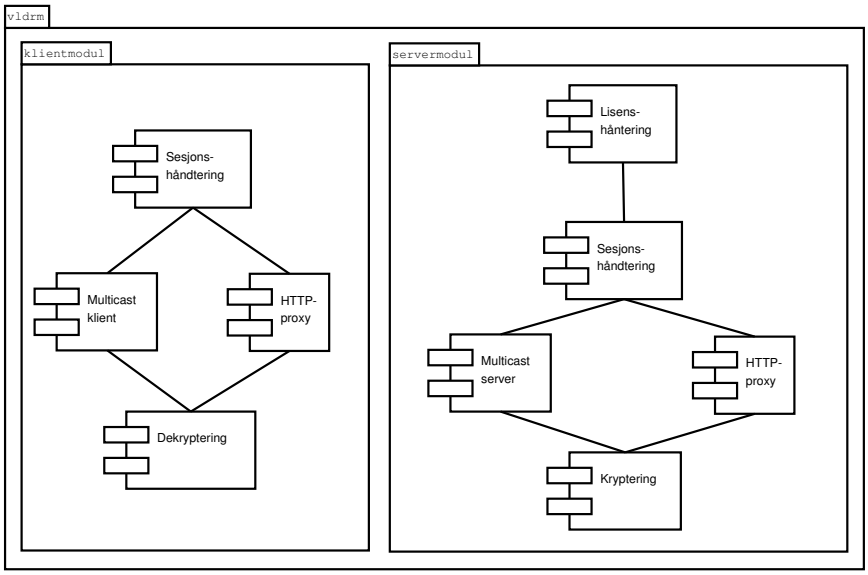
- Mediaserver (streamingserver). En form for kildesignal med video som skal krypteres. I eksemplene våre brukes VideoLAN (se avsnitt 5.2.1) for å generere et kildesignal fra en statisk MPEG-fil.
- Servermodul (*vldrmd*[1]). Krypterer multicast- og unicast-strømmer, autentiserer klientmoduler og distribuerer krypteringsnøkler.
- Avspiller (hos klient). Dette er for eksempel VideoLAN videoavspilleren (se avsnitt 5.2.1).

[1] *Vldrm(d)* er en sammensetning av *VideoLAN* (siden denne brukes til å teste prototypen), DRM (for hvilken funksjon komponentene utfører, og *d*-en betegner komponenten som en *daemon*, altså UNIX-verdenens betegnelse på en serverapplikasjon.

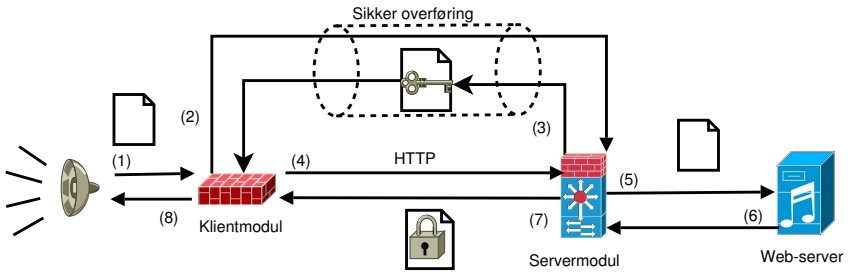
- Klientmodul (*vldrm*). Avspilleren bruker denne som en proxy for å få tilgang til kryptert innhold. Her blir dataene dekryptert og videresendt til avspilleren.

Det som kan gjøre terminologien litt komplisert er at begge disse (både server og klient) igjen fungerer som server og klient for henholdsvis media-avspiller og mediaserver. Men vi holder oss til å bruke betegnelsen *klient* om alle komponentene som befinner seg i avspillerenden av systemet. Det vil si avspilleren og klientmodulen som er tenkt å kjøre i en set-top-boks. Og betegnelsen *server* brukes i andre enden, om mediaserver (streaming-server) og servermodul.

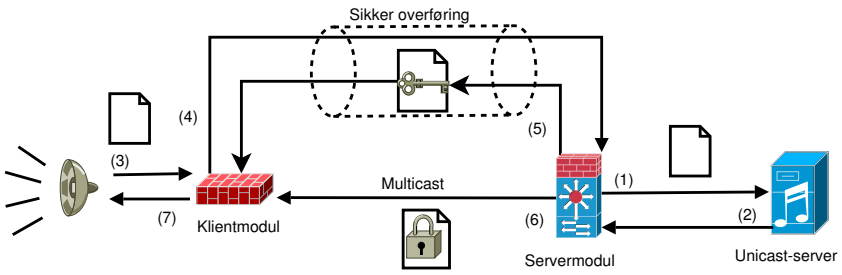
Figur 4.1 viser noe av de indre forholdene i våre to komponenter, mens figurene 4.2 og 4.3 også viser de eksterne komponentene og sekvensflyt av media- og kontrolldata gjennom systemet.



Figur 4.1: Systemkomponenter



Figur 4.2: Dataflyt for unicast-beskyttelse



Figur 4.3: Dataflyt for multicast-beskyttelse

4.1.1 Klientmodul

Klientmodulen har som hovedoppgave å sende forespørsel etter data, motta krypterte data, dekryptere data og videresende til avspiller. Den skal kun håndtere en datastrøm av gangen, og har slikt sett en enkel kontrollflyt. Valg av struktur og del-komponenter i klientmodulen er tatt med hensyn til begrensede ressurser på maskinvaren som komponenten skal kunne kjøres på. Dette innebærer blant annet:

- Minimal bruk av store programbiblioteker. Klientmodulen er i hovedsak lenket til bibliotek for krypteringsfunksjonalitet.

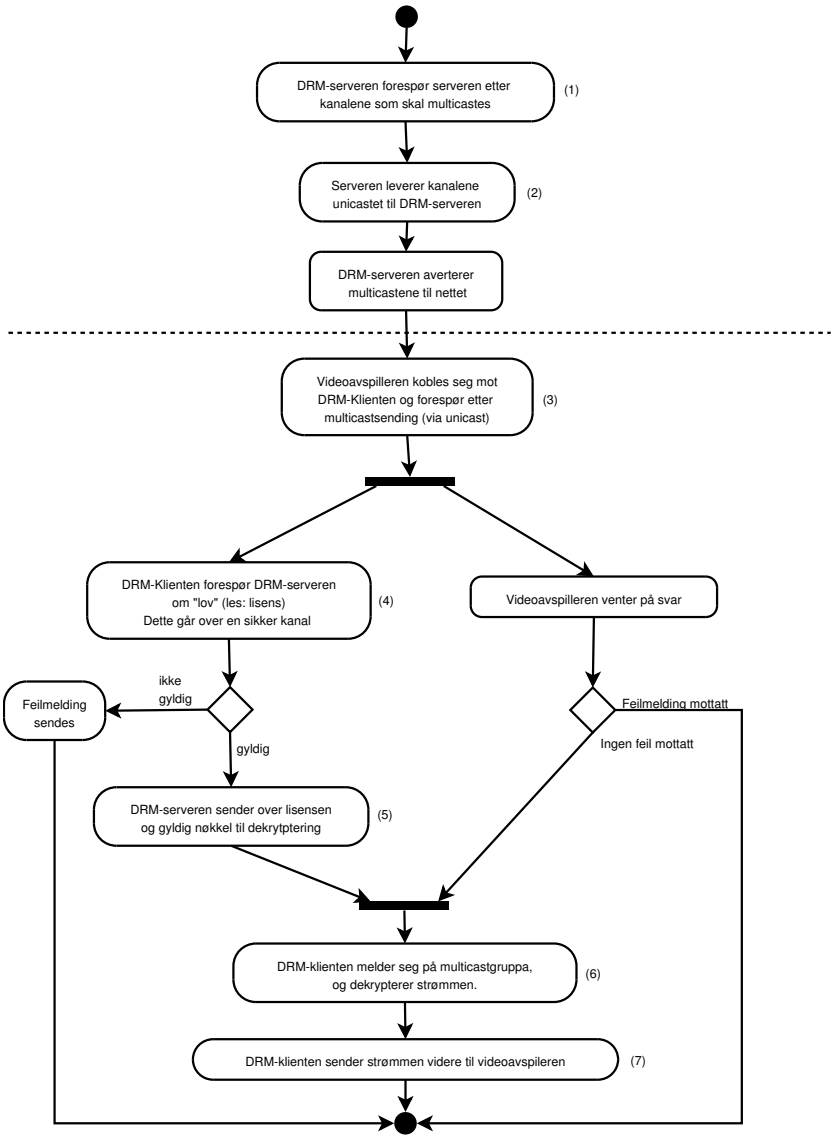
- En-trådet. Innhenting av krypteringsnøkler er et eksempel på funksjonalitet som eventuelt kan kjøres i en egen tråd. Ved å holde programmet i én prosess slipper man kontekstbytte i operativsystemet internt i programmet. Kontekstbytte vil fremdeles måtte skje mellom klientmodul og avspiller, siden disse kjører på samme prosessor.

Multicast på klientmodul

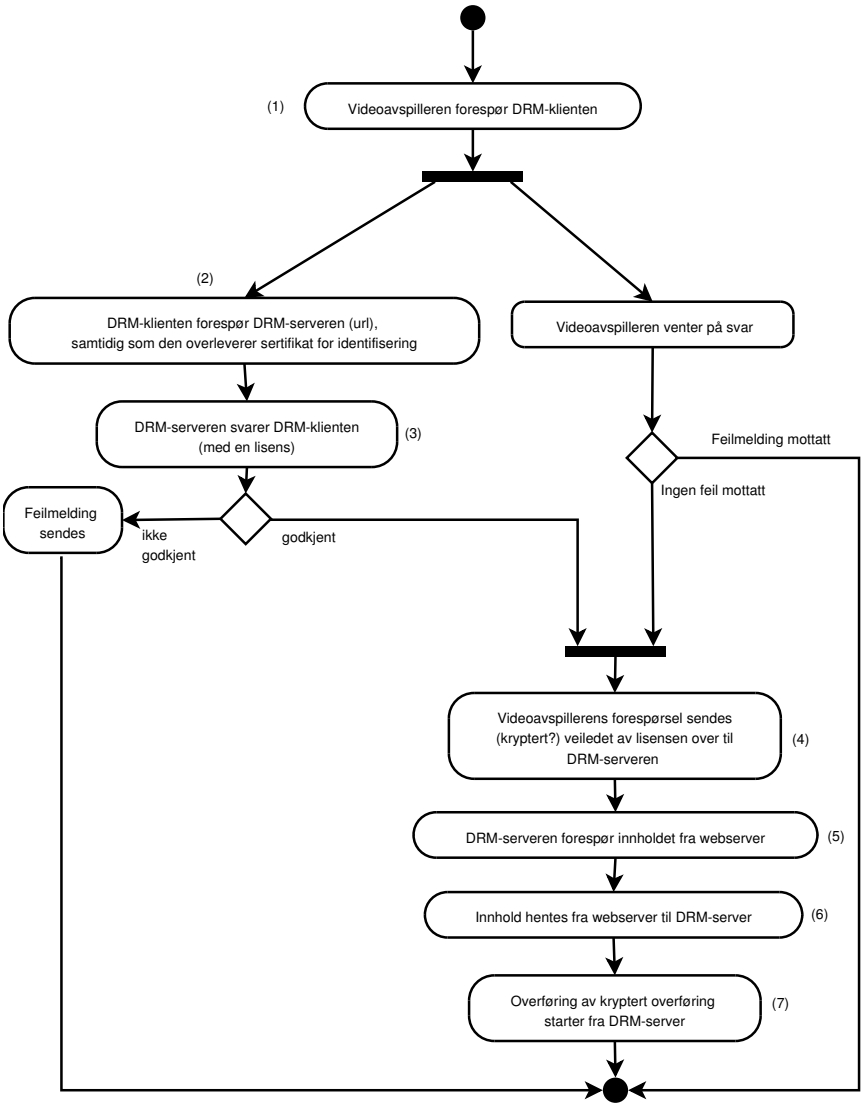
Klientmodulen har som hovedoppgave å dekryptere multicast-data. Tallene som er brukt her referer til figurene 4.3 og 4.4. Klientmodulen kobler seg til servermodulen via en sikker kanal (4). Servermodulen kontrollerer at klienten har lov til å motta multicast-sendingen. Hvis klienten er godkjent mottar den en dekrypteringsnøkkel til multicast-sendingen (5). Klientmodulen melder seg deretter på multicast-gruppen og dekryptere strømmen (6).

Unicast på klientmodul

Unicast-delen av klienten vil ha som hovedoppgave å dekryptere HTTP-data som kommer fra serveren. Hele klientmodulen fungerer som en HTTP-proxy, eller et ekstra ledd i en vanlig HTTP forespørsel. Dette er illustrert i figurene 4.2 og 4.5.



Figur 4.4: Aktivitetsdiagram for multicast-beskyttelse



Figur 4.5: Aktivitetsdiagram for unicast-beskyttelse

4.1.2 Servermodul

Servermodulen har som hovedoppgave å motta forespørsler, kontrollere forespørselen, kryptere forespurt data og sende dette til klientmodulen. Serveren har en mer kompleks multi-trådet oppbygning med en tråd for hver multicast-kanal og en tråd for hver kontrollkanal til klientene. I tillegg til kontrolltrådene som driver serversystemet.

Multicast fra servermodul

Alle multicast-strømmene blir sendt som unicast-strømmer fra media-server til servermodulen. Her blir strømmene kryptert og sendt ut som multicast-strømmer. Servermodulen mottar forespørsler etter krypteringsnøkkelen over en sikker kanal. Det detaljerte forløpet er vist i figurene 4.3 og 4.4.

Unicast fra servermodul

Unicast serveren virker kun som et ekstra ledd i HTTP-forespørselen, som har som oppgave å kryptere den returnerte media-strømmen. Individuell nøkkel for dekryptering leveres fra servermodulen over en sikker kanal hvis klienten har tilgang til datastrømmen. Dette kommer frem i figurene 4.2 og 4.5.

4.2 Tilgangskontroll

Tilgangskontrollen er lagt opp til å være basert på en unik ID i sertifikatet til klienten. Denne IDen kontrolleres mot en brukerdatabase for kontroll av tilgang. Autentiseringen av en klient vil da basere seg på gyldigheten av det signerte sertifikatet.

Brukerdatabasen må ha informasjon om hvilke rettigheter de forskjellige brukerne har til å aksessere tilgjengelige objekter. Grensesnitt mot denne brukerdatabasen er ikke definert, men bør for eksempel la seg integrere mot eksisterende betalingsløsninger. Tilgangsinformasjonen til enkelte objekter, eller grupper av objekter, bør utformes i henhold til en definert standard for beskrivelse av mediaobjekter, for eksempel MPEG-REL. En betalingsløsning eller annet administrativt system vil generere lisenser (eller lisensinformasjon) som lagres sentralt.

Del III

Utvikling

Kapittel 5

Verktøy

Dette kapittelet gir en oversikt over hvilke verktøy vi har brukt til utvikling og testing, samt hvordan vi har tatt i bruk noen av disse.

5.1 Programmeringsverktøy

5.1.1 Utviklingsplattform

Utviklingen av prototypesystemet beskrevet i denne rapporten har foregått i operativsystemet Linux. Dette ble valgt delvis fordi flere set-top-bokser er basert på Linux, og fordi SnapTV har fra før tatt i bruk *open source*-biblioteker i sine løsninger. En annen grunn til å velge Linux som utviklingsplattform var for å dra nytte av de verktøy og programbiblioteker som er fritt tilgjengelige for denne plattformen. I tillegg er dette en plattform vi er komfortable med fra før.

5.1.2 Programmeringsspråk

Vårt valg av programmeringsspråk falt på C++. For vår egen og oppgavens del, ville Java vært det enkleste valget for å implementere en prototype. I utgangspunktet burde klientmodulen utvikles i C da denne potensielt skal kjøres på en set-top-boks der datakraften og ressursene er knappe. Som et kompromiss valgte vi C++ for å kunne bruke samme programmeringsspråk på både klient- og server-modul. Når grensesnittet mellom klient og server derimot er implementert, vil det ikke være noe i veien for å implementere noen av modulene i et annet språk om det trengs.

5.1.3 Debugging

The GNU Project Debugger[1] (GDB) ble brukt til debugging. GDB er en debugger for C og C++ programmer. Programmet man ønsker å debugge

[1] GDB: http://www.gnu.org/software/gdb/

startes opp ved hjelp av GDB. Hvis det så oppstår en feil under kjøring av programmet kan man undersøke dette nærmere. Ved hjelp av GDB kan man da i detalj undersøke hvilke metoder som ble eksekvert rett før et fatalt signal, undersøke minne og programregistre, samt endre programmet for å forsøke å rette feilen.

5.1.4 Testrammeverk

Det er flere tilgjengelige rammeverk for enhetstesting i C++. Valget vårt falt på TUT[2]. Dette rammeverket ble i utgangspunktet valgt fordi det ikke er avhengig av eksterne programbibliotek, men er programmert i STL[3]-stil og består av kun en header-fil. Dette gjør også at testrammeverket er mer portabelt til andre utviklingsplattformer. Vårt første forsøk var med CPPUnit, som er basert på funksjonaliteten til JUnit for Java. Dette fungerte bra, men hadde ulempen av å måtte installeres eksplisitt på maskinen som skal kjøre testene.

5.1.5 Valgrind

Valgrind er et program for å sjekke minneaksesseringsfeil. Til forskjell fra andre minneaksesseringsverktøy, som trenger å lenke inn egne versjoner av for eksempel *malloc()*, *free()*, *new* eller *delete* for å holde rede på minneaksess og allokering, starter Valgrind programmet i en egen virtuell maskin. Fordelen med dette er at Valgrind får kontroll over alt programmet foretar seg, og kan derfor detektere flere typer minneaksesseringsfeil. Ulempen er at programmet kan bli opptil 50% så tregt og kan derfor oppføre seg annerledes enn ved en vanlig kjøring.

5.2 Programbiblioteker og andre verktøy

I utviklingen av prototypen vår har vi forsøkt å benytte oss av ferdige programbiblioteker der det er tilgjengelig for funksjonaliteten vi har hatt brukt for. Dette gjelder spesielt krypteringsfunksjonalitet. Av andre verktøy er det VideoLAN som har vært mest til nytte under hele prosessen, både for eksperimentering før og under utviklingen, og til testing av prototypen.

5.2.1 VideoLAN

VideoLAN[4] er et åpent kildekode-prosjekt for streaming og avspilling av multimediadata, i hovedsak video. Til å begynne med besto prosjektet av en streaming-server (vls) og en avspiller (vlc). Avspilleren har nå etterhvert

[2]TUT: http://tut-framework.sourceforge.net/
[3]standard template library
[4]VideoLAN: http://www.videolan.org/

også fått streaming-funksjonalitet og *transcode-mulighet*[5]. VideoLAN støtter et stort antall video-formater, som blant annet: MPEG-1, MPEG-2, MPEG-4, DivX, DVD, digitale satellitt kanaler (DVB). Streaming over IP-versjon 4 og 6 er støttet, både via unicast og multicast. VideoLAN støtter svært mange plattformer: GNU/Linux varianter, BSD, Windows, Mac OS X, BeOS, Solaris og QNX. Funksjonaliteten er størst under GNU/Linux. VideoLAN-prosjektet startet som et studentprosjekt ved en ingeniørhøgskole i Frankrike, men er i dag et verdensomspennende prosjekt hvor alle kan bidra.

Vi har brukt VideoLAN som både avspiller og streaming-server. Dette gjør også SnapTV i sine prototyper. Systemet vårt er ikke avhengig av VideoLAN, men det er dette programmet vi i størst grad har benyttet til testing. Det er svært tolerant ved feil i strømmen, og bruker minimalt med buffring i avspilling, noe som gjør det ideelt til testformål.

5.2.2 GnuTLS

Det er hovedsakelig to programbiblioteker som benyttes for å implementere TLS og SSL; henholdsvis OpenSSL[6] og GnuTLS[7]. Selv om navnet på bibliotekene kan forvirre, så støtter begge bibliotekene SSL og TLS, samt annen funksjonalitet vi har behov for.

TLS (Transport Layer Security) har overtatt etter SSL, og brukes til å etablere sikre kommunikasjonskanaler. SSL-protokollen ble i sin tid utviklet av Netscape for å kryptere HTTP-forbindelser. Protokollen kan også brukes til å autentisere endepunktene.

GnuTLS støtter følgende funksjonalitet:

- TLS 1.0/1.1 og SSL 3.0
- X.509 og OpenPGP sertifikater (håndtering, verifisering og autentisering)

Vi har valgt å benytte GnuTLS til å implementere en sikker kanal mellom server- og klientmodulen, hvor vi både autentiserer klienten, overfører nøkler og kontrollinformasjon.

5.2.3 gcrypt

Gcrypt[8] er et generelt krypteringsbibliotek. Det støtter symmetriske- og asymmetriske chiffere, hash algoritmer, meldingsautentisering, MPI[9] og generering av slumptall. Programbiblioteket baserer seg på kildekoden til GnuPG. Det støtter de fleste vanlige algoritmer og har mulighet for å registrere

[5] Transcode: Omkoding av strømmen i sanntid.

[6] OpenSSL: http://www.openssl.org

[7] GnuTLS: http://www.gnu.org/software/gnutls/

[8] gcrypt: http://directory.fsf.org/security/libgcrypt.html

[9] Multiple Precision Integers (MPI), datatyper og aritmetiske funksjoner for heltall større enn vanlige 32 bits heltall.

andre algoritmer hvis det er ønskelig. Denne muligheten gjør at man for eksempel kan skifte ut et symmetrisk chiffer med et som er raskere, sikrere eller utbedrer av en bestemt svakhet.

Vi bruker dette biblioteket til kryptering av media-strømmen. Gcrypt er skrevet i C, men kan brukes fra C++.

5.2.4 libxml

Det finnes flere tilgjengelige multiplattform-programbiblioteker basert på åpen kildekode for XML-parsing. De mest aktuelle som er tilgjengelige for C/C++ er *expat* (fra Mozilla), *libxml* (fra Gnome) og *Xerces C* (fra Apache). Våre kriterier for valg av bibliotek var at vi trengte relativt enkel parser- og valideringsfunksjonalitet. *XML Benchmark*[10] er et prosjekt som sammenligner funksjonalitet og ytelse til flere XML-parsere. Xerces-biblioteket er, i motsetning til *libxml* og *expat*, objektorientert. Men basert på resultatene til *XML Benchmark* er dette stort og tregt, med mer funksjonalitet enn vi har brukt for. *Expat* er på den andre siden for enkelt og mangler blant annet valideringsfunksjonalitet. Ut i fra ytelse, størrelse og funksjonalitet valgte vi derfor *libxml* som passet våre behov.

Libxml[11] er i utgangspunktet XML-programbiblioteket som brukes i Gnome-prosjektet[12]. Selv om Gnome-prosjektet er avhengig av dette programbiblioteket er ikke *libxml* avhengig av Gnome.

Opprinnelig var tanken at XML skulle brukes til å beskrive innholdslisensene i systemet. Som vi beskriver senere i rapporten falt denne bruken bort, i forbindelse med prototypen. Hovedbruken av XML i vårt system går på lesing, skriving og validering av konfigurasjonfiler. Ved å bruke XML til dette formålet kunne vi dra nytte at XML-bibliotekets parsing- og valideringsfunksjonalitet uten å bruke tid på å utvikle kode for dette. Det var da mulig å beskrive konfigurasjonen ved hjelp av XML schema slik at dette senere kan valideres før applikasjonen leser konfigurasjonen.

5.2.5 GNU Common C++

GNU Common C++[13] (libcommoncpp) er en samling med C++ klasser som abstraherer en del systemtjenester. Bruken av dette biblioteket gjør kildekoden mer portabel til andre plattformer. Det støtter tjenester for flertrådsprogrammering, synkronisering, nettverktilgang og annet.

Hovedbegrunnelsen for å bruke dette biblioteket var på grunn av *Thread*-klassen, som implementerer POSIX-threads på en måte som minner om trådprogrammering i Java, samt klassen for mutex-håndtering.

[10] XML Benchmark: http://xmlbench.sourceforge.net
[11] libxml: http://xmlsoft.org
[12] Gnome: http://www.gnome.org
[13] GNU Common C++: http://directory.fsf.org/GNU/commoncpp.html

5.2.6 CVS

Concurrent Versioning System (CVS) er et uvurderlig verktøy i et utviklingsprosjekt med mer enn en utvikler. Kildekoden lagres sentralt på en maskin, og hver utvikler sjekker ut en arbeidskopi til en lokal maskin og gjør endringer der før det sjekkes inn igjen til det sentrale lageret. Slik jobber utviklerne med samme kodebase, eller til og med samme fil. Er det gjort endringer i samme fil, vil logikken i programmet flette sammen endringene ved innsjekking. Hvis endringene er for nær hverandre i en kildefil, vil konflikter kunne oppstå som man må ordne opp i manuelt.

Den andre fordelen ved å bruke CVS er (som navnet tilsier) versjonskontroll. Alle endringer som sjekkes inn lagres. Man kan hente ut eldre versjoner av en fil, hvis man for eksempel er interessert i å se hva som er endret i forbindelse med en feil i programmet.

5.2.7 Doxygen

Doxygen[14] kan sammenlignes med JavaDoc som brukes til å generere dokumentasjon ut i fra kildekoden. Kommentarene i koden formateres med en spesiell syntaks slik at klasser, metoder og attributter forbindes med den gitte kommentaren. Dette kan så brukes til å generere dokumentasjon i HTML, RTF eller LaTeX.

5.2.8 autoconf/automake

Autoconf og *automake* er to verktøy som ofte brukes i prosjekter som distribueres som kildekode. Det forenkler tilpasning av *makefiler* til forskjellige systemkonfigurasjoner. *Autoconf* brukes til å generere et konfigurasjonsskript ut i fra en beskrivelse av programmer og betingelser man ønsker å teste for lokasjon og eksistens. Filen `configure.ac` brukes til å beskrive et sett med betingelser, og *autoconf* generer skriptet *configure* som gjør selve jobben. Dette skriptet kan også brukes til å substituere verdier i for eksempel header-filer, dokumentasjonsfiler eller andre tekstfiler.

Automake generer en eller flere *makefiler* ut i fra en beskrivelse i filen *Makefile.am*. *Makefilene* beskriver hvordan programmet skal kompileres og avhengighetene mellom modulene i programmet.

5.2.9 CCCC

CCCC[15] (C and C++ Code Counter) er et verktøy for å analysere visse egenskaper ved kildekode i C/C++. Fra forholdet mellom antall kodelinjer og kommentarer, til objektorienterte og strukturelle egenskaper.

[14] Doxygen: http://www.doxygen.org
[15] CCCC: http://cccc.sourceforge.net/

5.2.10 tcpdump

Tcpdump er et verktøy for å lagre nettverkstrafikk for senere analyse. Dette kan være nyttig hvis man ønsker å se nærmere på en nettverksprotokoll og de individuelle pakkene som nettverkskortet i maskinen plukker opp fra nettverket.

5.2.11 ffmpeg

Ffmpeg[16] er en samling av programbiblioteker og verktøy for å spille av, konvertere og streame forskjellige audio og video formater. Vår interesse i dette prosjektet er programbibliotekene libavcodec og libavformat som kan integreres i koden for å tolke innholdet i for eksempel MPEG-filer, samt eksperimenter med streaming og for å få opplysninger om filformater brukt i tester og eksperimenter.

Grunnen til at det er to biblioteker (codec og format) er fordi de forskjellige video-codecene ofte er pakket inn i et generelt format. Eksempler på dette kan være Quicktime-, OGG- eller AVI-formatet som igjen kan inneholde flere strømmer med forskjellige audio- og video-codecs. Libavformat inneholder derfor kode for å tolke metaformatene og libavcodec tolker de forskjellige codecene inne i disse formatene. Figur 5.1 viser de forskjellige strømmene som finnes inne i en MPEG PS-formatert fil. Her ser vi at formatet er bygget opp av en videostrøm som bruker *mpeg2video*-codecen og en audiostrøm som bruker *mp2*-codecen. Programmet *ffprobe* er forøvrig et testprogram vi lagde da vi gjorde oss kjent med APIet i programbibliotekene. Et nærmest identisk resultat kan fås med *-stats*-parameteret til *ffplay*-programmet.

```
$ ffprobe stallman_oslo_20040916_1.mpeg
mpeg - MPEG PS format
Input #0, mpeg, from 'stallman_oslo_20040916_1.mpeg':
Duration: 00:04:32.8, start: 0.000000, bitrate: 299 kb/s
Stream #0.0: Video: mpeg1video, 384x288, 25.00 fps
Stream #0.1: Audio: mp2, 32000 Hz, mono, 48 kb/s
```

Figur 5.1: Innholdet i en MPEG PS-fil.

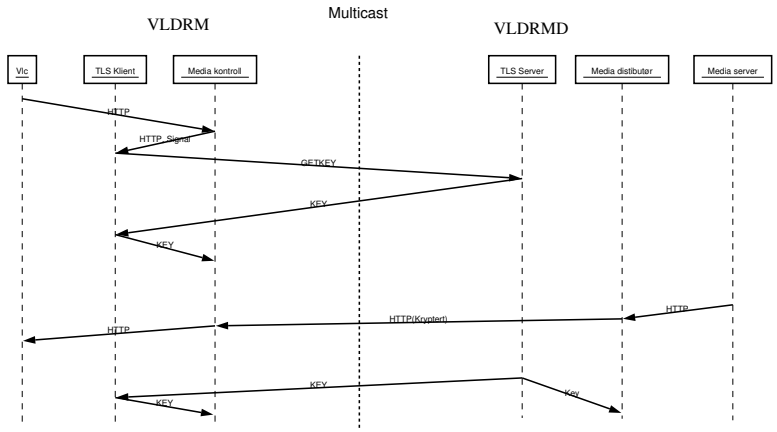
[16] ffmpeg: http://ffmpeg.sf.net

Kapittel 6

Implementasjon

Kapittelet vil beskrive hvordan vi løste selve implementasjonen på et mer detaljert nivå enn i designbeskrivelsen. Vi legger vekt på det som er spesielt i løsningen vår, og de punktene vi har valgt *originale* løsninger.

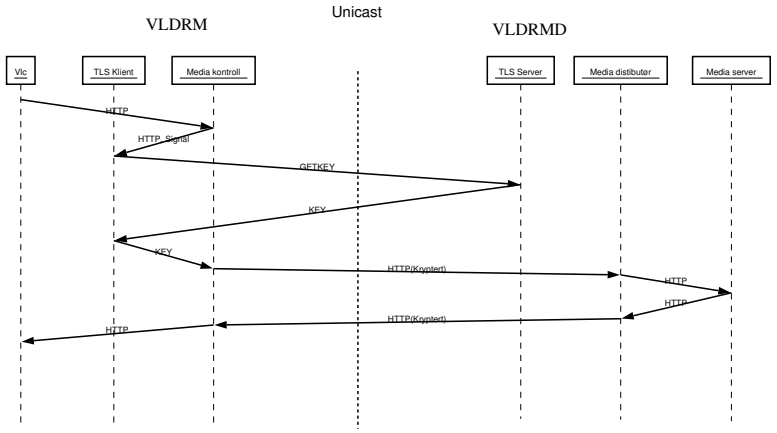
6.1 Kommunikasjon



Figur 6.1: Sekvensdiagram for ekstern kommunikasjon ved multicast

6.1.1 Serverkommunikasjon

Kommunikasjonen mellom klient- og servermodulen skjer over en sikker kanal (TLS). Denne bygger på sertifikatautentisering av bruker og kontroll av at brukerens sertifikat, sertifikatsignatur og navn er godkjent av serveren. Vi benytter X.509-sertifikater til autentisering av klienter i systemet. Opprettelse av sikker kanal og sertifikatautentisering benytter funksjonalitet i GnuTLS-biblioteket (se avsnitt 5.2.2). På figuerene 6.2 og 6.1 er all



Figur 6.2: Sekvensdiagram for ekstern kommunikasjon ved unicast

kommunikasjon med komponenter med TLS i navnet kryptert. Serveren får oversendt sertifikatet som kontrollerer at det er signert av CAen til serveren. Navnet (DN i X.509) eller annen informasjon om brukeren kan så kontrolleres opp mot en database, konfigurasjonsfiler eller andre former for tilgangskontroll. Da dette er kontrollert og *handshake* mellom klient og server er ferdig, har disse opprettet en sikker forbindelse mellom server og klient.

Over denne sikre forbindelsen sender så klienten sin forespørsel etter en dekrypteringsnøkkel. Denne forespørselen er en formatert tekststreng som identifiserer objektet som det ønskes tilgang til. Formatet til denne tekststrengen er som beskrevet i avsnitt 6.1.2. Det vil si at for en forespørsel etter en unicast-strøm vil hele URLen overføres som en objekt-ID, for en multicast-strøm vil gruppeadresse og portnummer overføres.

Dette legges til grunn for hvilken nøkkel som skal sendes tilbake til klientmodulen over den sikre kanalen, eller eventuelt avvise forespørselen hvis denne brukeren ikke har tilgang.

Nøkkelutvekslingsformat

Linjen nedenfor viser formatet en krypteringsnøkkel blir overført fra servermodul til klientmodul.

```
KEY cipher mode id keylen noncelen keydata nextid nextkeylen \
nextnoncelen nextkeydata
```

Feltbeskrivelse:

- *cipher*. Chifferalgoritme brukt i krypteringen. Alle chiffer støttet av *gcrypt* kan brukes. For eksempel AES eller Twofish.
- *mode*. Chiffermodus. CTR er standard, men CFB kan også benyttes. ECB og CBC kan ikke benyttes da det ikke er implementer padding av data.

- *id*. Nøkkel-ID. Heltall.
- *keylen*. Nøkkellengde i bytes. Standard for AES er 16 bytes.
- *noncelen*. Lengde på *nonce* for å regenerere IVen. *Nonce*-dataene er de siste *noncelen* bytes i *keydata*.
- *keydata*. Krypteringsnøkkelen og eventuelt en *nonce* hvis *noncelen* er større enn 0. Dette er binærdata som er kodet i *base 64*.
- *nextid*. Nøkkel-ID for krypteringsnøkkelen som vil bli benyttet ved neste nøkkelskifte.
- *nextkeylen*. Nøkkellengde.
- *nextnoncelen*. *Nonce*-lengde.
- *nextkeydata*. *Base 64*-kodet nøkkeldata.

De fire siste feltene kan ha verdien 0. Dette vil være tilfellet første gang klientmodulen får overført en nøkkel. Ved nøkkelskift vil disse feltene inneholde data om nøkkelen som vil være aktiv etter gjeldene nøkkel.

I tekststrengen under vises et eksempel på nøkkelutvekslingsformatet. Her dataene kryptert med AES i CTR-modus og nøkkellengden er 16 bytes med en 4 byte *nonce*. Deretter følger *base 64-kodet* nøkkeldata. Feltene for nøkkelskifte er 0.

```
KEY AES CTR 123 16 4 MDEyMzQ1Njc4OUFCQ0RFRkdISUo= 0 0 0 0
```

Nøkkelbytte

Ved multicast kreves nøkkelbytte, se avsnitt 6.2.2 for hvorfor. Hver enkelt multicast-tråd er ansvarlig for å generere en ny krypteringsnøkkel når dette er nødvendig eller definert etter et visst intervall. En egen tråd kjører parallelt med disse trådene og sjekker når det er signalisert et nøkkelbytte. Kall dette gjerne *nøkkelbyttetråden*. Denne tråden går så i tur og orden gjennom alle multicast-tråder som har signalisert et nøkkelbytte. Hver multicast-tråd har en liste med aktive klienter, så alle disse vil få tilsendt den nye nøkkelen. Nøkkelbyttetråden signaliserer så tilbake til multicast-tråden at alle klientene har mottatt den nye nøkkelen, og den nye nøkkelen kan aktiviseres.

Klienter som kobler seg på midt i et nøkkelbytte vil få tildelt både gjeldene og neste krypteringsnøkkel ved tilkoblingstidspunktet.

Alle trådene er asynkrone, men grensesnittet mellom multicast-trådene og nøkkelbyttetråden er synkronisert ved å bruke en *mutex*.

6.1.2 Lokal klientkommunikasjon

Siden vi har valgt en løsning hvor klientmodulen er uavhengig av avspilleren trenger vi en form for kommunikasjon mellom disse to komponentene (se punkt 3 og 7 i figur 4.3). Alternativet ville vært en form for plug-in-løsning hvor dekrypteringsfunksjonaliteten var bygd inn i avspilleren. Det vi trengte var en utbredt protokoll som støttes av de fleste avspillere. Vi har også fordelen med at denne kommunikasjonen vil foregå lokalt på maskinen. Vårt valg falt på HTTP-protokollen. Vi har basert dette valget på at dette er en enkelt protokoll å implementere, samt at de fleste avspillere støtter streaming av video fra en HTTP-server. I utgangspunktet er ikke HTTP en like bra egnet protokoll som for eksempel RTP. HTTP bruker TCP som transportprotokoll, mens RTP bruker UDP. Vi mener likevel at dette i vårt tilfelle er en protokoll som er i stand til å gjøre jobben. Spesielt siden TCP-forbindelsen vil være lokal, og derfor vil ha høyere ytelse enn over et nettverk. Se vedlegg C.2 for en ytelsestest på en lokal TCP-forbindelse.

Avspilleren vil da se klienten som en HTTP-server. Vi har da to tilfeller for hvordan en mediastrøm skal adresseres:

- Unicast. For unicast-strømmer vil klientmodulen fungere som enn HTTP-proxy. Det vil si at den videresender forespørselen til servermodulen for å koble seg til en kryptert strøm.

  Format på en unicast-forespørsel fra avspilleren:

  `http://<local adresse>/<url>`

  Eksempel på en multicast-forespørsel:

  `http://127.0.0.1/bruker1/starwars_episode1.mpeg`

- Multicast. For å koble seg på en multicast-strøm bruker avspilleren en URL med en spesiell syntaks som vist under. Klientmodulen melder seg på multicast-gruppen som spesifisert i URLen og videresender multicast-pakker som HTTP-trafikk til avspilleren som vist i figuren 6.1.

  Adresser i området 232/8 behandles som SSM-adresser, resten av adresseområdet 224/4 benytter ASM (se avsnitt 2.6.1).

  Format på en multicast-forespørsel fra avspilleren:

  `http://<local adresse>/MULTICAST/<multicast gruppe>:<port>`

  Eksempel på en multicast-forespørsel:

  `http://127.0.0.1/MULTICAST/232.0.0.1:1234`

HTTP-grensesnittet i klientmodulen er ikke en full implementasjon av RFC1945 (HTTP 1.0) eller RFC2616 (HTTP 1.1), men et minimalt grensesnitt for å kunne kommunisere med en avspiller som støtter HTTP-streaming.

6.2 Innholdskryptering

Når vi ser på pakkene fra VideoLAN ved multicast/unicast sendinger så har disse en størrelse på 1316 byte ved streaming i MPEG TS-format (ved bruk av tcpdump, se avsnitt 5.2.10). Denne størrelsen forklares med at VideoLAN sender 7 pakker av 188 bytes, hvor 188 er størrelsen på en TS-pakke i MPEG. Denne størrelsen er valgt for å holde seg nær MTU-størrelsen i et IP-nettverk. UDP/TCP- og IP-headere kommer i tillegg, og man holder seg under MTU som i ethernet er på 1500 bytes.

For kryptering med blokkchiffere er denne størrelsen ugunstig, da den ikke går opp i blokkstørrelsen som i nyere chifferalgoritmer er på 16 bytes. Ved kryptering av UDP-pakker må man da legge til data i hver pakke for å komme opp til nærmeste størrelse som går opp i blokkstørrelsen, som for VideoLAN blir 1328 bytes. Dette vil medføre økt båndbreddekrav i forhold til ukrypterte data. Chiffermodi som ECB og CBC har dette problemet. Chiffermodi som for eksempel CFB eller CTR fungerer som nevnt som strømchiffere og kan brukes direkte på uendret pakkestørrelse.

Ved bruk av CFB eller CTR slipper vi likevel ikke unna med å øke båndbredden som trengs for krypterte data i forhold til ukrypterte data. Hvis man med TCP skal overføre en fil over et nettverk kan dette gjøres i for eksempel CBC-modus. Initialiseringsvektoren (IV) til første blokk er det eneste man trenger å overføre. Ved kryptering av UDP-pakker krypterer man en og en melding av gangen og man vil trenge å overføre en IV for hver pakke.

6.2.1 Valg av krypteringsmetode

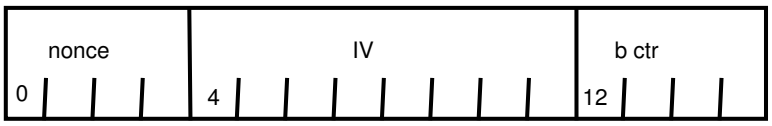
Vi har valgt å bruke AES i CTR-modus med 128 bits nøkler. Dette er implementert som beskrevet i [Hou04] og [All04]. CTR egner seg godt til denne type data hvor individuelle meldinger overføres. CTR er en anbefalt modus i blant annet [FS03] og brukes også i SRTP [BMN+04].

Figur 6.3 viser hvordan initialiseringsvektoren i CTR-modus er oppbygd. Feltene er beskrevet som følger:

- *nonce*. 4 byte. Denne verdien genereres som fire ekstra bytes når krypteringsnøkkelen genereres. Det vil si at for AES med en nøkkelstørrelse på 16 bytes, genereres 20 bytes. De 4 siste bytene brukes til denne verdien. Begrunnelsen for denne verdien er i følge [Hou04] for å sikre mot enkelte *precomputation attacks*.

- *iv*. 8 byte initialiseringsvektor. Denne velges tilfeldig for første pakke. Deretter økes den med 1 for hver melding. Den må være unik for hver nøkkel. Ved å la den være en teller er dette kriteriet oppfylt så lenge ikke telleren har gått rundt og tilbake til startverdien.

- *block counter*. 4 byte. For hvert kall til krypteringsmetoden vil denne starte på verdien 1 og inkrementeres for hver chifferblokk. Det vil si at meldinger opptil $2^{32} - 1$ bits kan krypteres. Verdien er definert i [Hou04] til å være big-endian, men siden vi behandler den som en serie på 4 bytes og ikke et heltall vil vi ikke trenge å ta noen spesielle hensyn til dette.



Figur 6.3: Format på IV i CTR-modus

For å bekrefte overensstemmelse med [Hou04] er testvektorene 1 til 3 fra dette dokumentet implementert i enhetstestrammeverket.

En mangel i vårt system er at vi utelater verifisering av integriteten av meldingene. [Hou04] anbefaler bruk av en form for meldingsverifikasjon, for eksempel HMAC.

Vedlegg E.1 og E.2 viser kildekode for hvordan henholdsvis klient- og servermodul tar i mot, krypterer og videresender data.

6.2.2 Nøkkelbytte

Det er to grunner til at systemet trenger nøkkelskifte i multicast-strømmen:

1. For å overholde begrensninger i hvor mange pakker som kan overføres med en nøkkel. Denne begrensningen er satt av IV på 8 bytes. Hver nøkkel kan maksimalt benyttes til å kryptere $2^{64} - 1$ meldinger.

2. Sikre at brukere som har sagt opp et abonnement på en kanal ikke forsatt skal kunne bruke sin gamle nøkkel til å få tilgang til kanalen. For å overholde dette kriteriet vil en multicast-strøm gjennomføre et nøkkelskifte minst hver annen time. Dette kan endres etter behov. Det kan for eksempel hende at man ikke har behov for å skifte nøkkel mer enn en gang om dagen.

Nøkkelskiftet gjennomføres da av systemet før en av disse kriteriene er oppfylt. Dette er også for å ta hensyn til videostrømmer med forskjellig bit-rate. Hvis man bare så på hvor mange pakker som ble sendt i en strøm vil en strøm med lav bit-rate nesten ikke ha nøkkelskift, mens i en strøm med høy bit-rate vil nøkkelskift forekomme ofte. I og med at tallet for hvor mange meldinger man kan sende før nøkkelen må byttes er såpass høyt, vil kriteriet for å ha en oppdatert nøkkel av abonnementshensyn stort sett være den avgjørende faktoren for hvor ofte et nøkkelskift gjennomføres.

For sømløst å få dette til må hver pakke identifisere hvilken nøkkel den har blitt kryptert med. Dette kan være en nøkkel-identifikator i form av et tall. Mottakeren må da allerede ha mottatt nøkkelen det skal skiftes til ved hjelp av en annen mekanisme og sjekke nøkkelidentifikatoren for når denne nøkkelen skal tas i bruk.

6.2.3 Meldingsnummerering

Videre må det tas hensyn til at UDP-pakker kan mottas i en annen rekkefølge enn de har blitt sendt ut i. En løsning kunne ha vært å legge til en buffer på klienten for så å forsøke og ordne om igjen på rekkefølgen. Men dette vil kreve mye logikk, minne og ikke minst ha potensialet til å ødelegge timingen på strømmen. Alternativet vi gikk for er å takle pakker som mottas ut av rekkefølge som pakketap. For å oppdage dette på klienten kan hver UDP-pakke nummereres. Hvis klienten mottar en UDP-pakke med et meldingsnummer som er lavere enn det forrige vil pakken kastes.

6.2.4 Meldingsformat

Figur 6.4 er et forslag til hvordan hver melding kan krypteres hver for seg ved å legge til en header foran hver krypterte datablokk. Headeren er på til sammen 16 bytes. Feltene er beskrevet som følger:

- *mid*. 4 byte meldingsidentifikator (little-endian). Øker med 1 for hver UDP-pakke. Brukes av klientmodulen til å sjekke for pakketap og rekkefølgen av mottatte pakker.

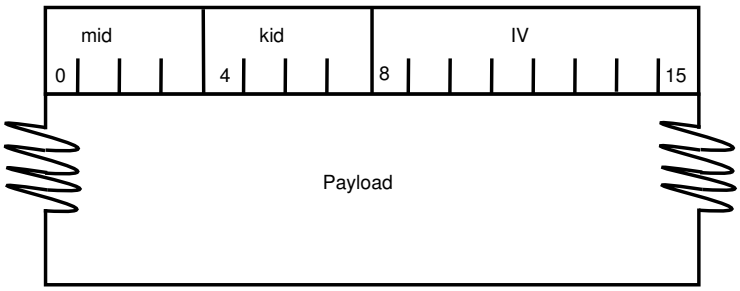
- *kid*. 4 byte nøkkel-identifikator (little-endian). Har verdien 0 for ukrypterte pakker. Starter på 1 og økes med 1 for hvert nøkkelbytte. Klientmodulen sjekker dette feltet og benytter seg av en krypteringsnøkkel med tilsvarende verdi som den skal ha fått tildelt på et tidligere tidspunkt.

- *IV*. 8 byte initialiseringsvektor. Disse 8 bytene er nok for klienten til å sette sammen en IV til dekryptering i CTR-modus.

Denne headeren blir en del av UDP-pakken sammen med de krypterte dataene. De to 32-bits tallene defineres for enkelhets skyld til å være i little-endian byte-rekkefølge. Implementeres løsningen på en big-endian maskin må dette tas hensyn til[1].

Vedlegg E.3 viser utdrag av kildekoden for formatering og distribuering av krypteringsnøkler ved multicast.

[1]Konfigurasjonsskriptet for applikasjonen detekterer om maskinen er big- eller little-endian.



Figur 6.4: Meldingsformat for kryptert multicast

6.3 Kildekode

Tabell 6.1 viser en oversikt over omfanget av kildekoden som er skrevet i forbindelse med dette prosjektet. Dette inkluderer ikke enhetstester. En mer detaljer analyse av kildekoden generert av verktøyet CCCC (se avsnitt 5.2.9) er å finne i vedlegg G.

Mål	Totalt	Per modul
Antall moduler	74	
Antall kodelinjer	6415	86,689
Antall kommentarlinjer	2821	38,122
Kodelinjer per kommentarlinje	2,274	

Tabell 6.1: Oppsummering av omfanget av kildekoden

6.4 Klassebeskrivelser

Dette avsnittet gir en kort beskrivelse av klassene i systemet. Se vedlegg D for klassediagrammene. Noen av klassene er felles for både klient- og servermodulen, og disse vil bli presentert først.

- *UdpReader* tilbyr funksjonalitet som leser en UDP-strøm. Denne socket-funksjonaliteten blir brukt på serveren til å lese unicast-dataene sendt fra mediaserveren. På klienten blir den bruk til å lese den krypterte multicast-strømmen. Fungere for UDP både i unicast og multicast.

- *Reader* er en abstrakt klasse som har virtuelle metoder for å lette implementasjonen av UdpReader. Reader klassene er beskrevet i klassediagram i avsnitt D.1.

- *Tls* er en basisklasse for å initialisere GnuTLS-biblioteket. Klasser som benytter seg av GnuTLS arver fra denne.

- *Config* er basisklasse som implementerer generelle funksjoner for all konfigurasjon på både server og klient.
- *Httpd* er en enkel HTTP-server, uten tråder og bruk av *fork*.

Krypteringsfunksjonaliteten er naturlig nok lik på både server- og klientsiden. Klassediagrammet i vedlegg D.1 viser mer detaljer her.

- *Gcrypt* er en basisklasse for å initialisere gcrypt-biblioteket.
- *Mpi* blir brukt til behandling av store helttall. Brukes i hovedsak til å inkrementere IVen i AES-CTR.
- *CipherKey* er klassen som tar seg av all nøkkelbehandling ut mot både klient og server. Objekter av dette inneholder krypteringsnøkkelen som blir brukt, og eventuelt neste nøkkel.
- *Cipher* er en klasse som gir et objektorientert grensesnitt til gcrypt.

6.4.1 Klasser i servermodulen

ChiperKey, *Chiper* og *UdpReader* spiller en stor rolle på serveren, diagrammet i D.2 viser denne sammenhengen. Resten av klassene i servermodulen vises i vedlegg D.2.

- *MulticastServer* er en trådklasse som startes opp for hver multicast-strøm som skal krypteres.
- *UdpWriter* håndterer utsending av unicast eller multicast UDP-datagrammer til nettverket.
- *McastKeyChanger* tar seg av alt som har med nøkkelbytte på de forskjellige multicast-serverene.
- *TlsServer* tar i mot forespørsler fra klientmodulen og starter en separat tråd for håndtering av sikker kommunikasjon til hver tilkobling.
- *TlsSession* håndterer sikker kommunikasjon mellom klient- og servermodul for tilgang til krypteringsnøkler.

Logg-funksjonaliteten til serveren er fordelt på tre klasser. Detaljer kan leses fra klassediagrammet i D.2.

- *Log* er en abstrakt klasse som har det som er felles for logg-funksjonaliteten.
- *LogSyslog* skriver logg-meldinger til systemloggen.
- *LogFile* skriver logg-meldinger til en fil eller skjerm (stderr).

Webserverfunksjonaliteten på serveren er beskrevet mer i detalj i klassediagrammet i D.2.

- *HttpdThread* utvider *Httpd*-klassen med flertrådsfunksjonalitet.
- *HttpdSessionThread* utvider *HttpdSession* og håndterer HTTP-tilkoblinger.
- *HttpdRequest* tolker og lagrer informasjon om HTTP-forespørselen.
- *Status* er en enkel HTTP-server som gir kontinuerlig status på multicast-sendingene over webgrensesnitt. Ikke ferdigstilt.
- *StatusSession* startes som en ny tråd for hver tilkobling til statusserveren.

Klassene som håndterer konfigurasjonsfilene. Sammenhengen med *Config* fra fellesklassene er illustrert i klassediagrammet i D.2:

- *XmlFile* generell XML-funksjonalitet. Blant annet validering mot DTD og XML Schema.
- *ConfigServer* leser og skriver konfigurasjonsfilen.
- *ConfigStream* tar vare på informasjon for hver enkelt multicast-strøm.
- *ConfigTarget* tar vare på informasjon for hver multicast-kilde og målstrøm.
- *ConfigClient* lagrer kommandolinjevalg for klienten.

Klassediagrammet for SQL-grensesnittet finnes i D.2.

- *Sql* er basisklassen for SQL-grensesnittet. Denne klassen er abstrakt for å støtte forskjellige SQL-implementasjoner.
- *SqlLite3* implementerer et grensesnitt mot SQLite (versjon 3). Ikke komplett.
- *SqlLite* implementerer et grensesnitt mot SQLite (versjon 2). Ikke komplett.

6.4.2 Klientmodulen

Klientmodulens klassediagram er å finne i D.3

- *HttpdSession* blir arvet av *MediaClientSession*. Inneholder funksjonalitet for å kontrollere tilkoblingen fra mediaavspilleren.
- *HttpRequest* håndterer parsing av HTTP forespørsler til klienten.
- *MediaClient* tar imot forespørselen fra mediaavspilleren. Denne klassen arver fra *Httpd*.

- *MediaClientSession* startes for hver tilkobling til *MediaClient* fra mediaavspilleren. Denne klassen arver fra *HttpdSession*
- *ClientSession* implementerer klientfunksjonaliteten for oppkobling av en sikker kanal og mottak av krypteringsnøkler.

Kapittel 7

Testing

Dette kapittelet er todelt. Vi gjør først en kort oppsummering av generelle metoder for programvaretesting. Vi går så gjennom en del praktiske tester av det utviklede systemet. I denne delen vil vi få bekreftet funksjonaliteten til programmet, samt undersøkt praktiske ressurskrav.

7.1 Programvaretesting

Programvaretesting er med på å sikre kvalitet, stabilitet og funksjonalitet i programmer [Sof].

- Korrekt og planlagt funksjonalitet.
- Optimal ytelse.

Til hovedformer for testing:

- *White box*. Dette er i hovedsak strukturell testing som blir utført ved forskjellige former for kodegjennomgang.
- *Black box*. Black box test skiller seg fra white box ved at funksjonalitet testes *uten kunnskap* om hvordan koden i komponenten løser oppgaven. Metoder som brukes her er observasjon av programforløp, enhetstesting, funksjonstest, destruktiv test og valideringstest. Funksjonstesten og valideringstesten er basert på at bestemt input skal gi forventet resultat. Destruktiv testing skjer ved å få programmet i situasjoner som helst ikke skal oppstå.

  Systemtest er en viktig del av dette hvor man tester hele systemet. Innenfor dette er ytelsestest-test (*preformance*/stresstest) et punkt. Systemet skal her utsettes for ekstreme mengder data/tilkoblinger for å se hvordan programmet reagerer på dette.

7.1.1 Enhetstesting

Enhetstesting er en metode for å teste funksjonaliteten og korrektheten til enkelte moduler og funksjoner. Det utvikles da i utgangspunktet en separat test for hver metode i en klasse. Dette isolerer de enkelte programdelene og viser at hver enkelt del fungerer korrekt. Dette har to formål [Wikc]:

1. Oppfordre til endringer (refaktorisere), lett å sjekke at ting fortsatt fungerer hvis kildekoden endres.

2. Fjerne usikkerhet rundt funksjonaliteten til enkelte programdeler.

Vi har ikke fulgt noen spesiell utviklingsmetodikk i vårt prosjekt, men enhetstesting er en av karakteristikkene i eXtreme programming (XP). Ideelt sett lager man testene først, det vil si at man definerer grensesnittet, og så implementerer funksjonaliteten. Slik vil alle testene feile helt til man har implementert funksjonaliteten korrekt.

En variant av enhetstesting er regresjons-testing. Her vil det opprettes tester når det oppdages en feil i programkoden. Testen vil implementeres før feilen rettes slik at man får verifisert at testen fanger opp feilen. Disse testene vil da være et sikkerhetsnett som fanger opp om feilen skulle bli gjeninnført i koden, for eksempel ved redesign eller vedlikehold.

Steget videre fra enhetstesting er integrasjonstesting. Her tester man hele grensesnittet til enkelte moduler og om interaksjonen mellom modulene er korrekt. For vårt tilfelle var dette for eksempel testing av klasser for UDP-kommunikasjon, lesing av konfigurasjonsfiler og lignende.

Refaktorisering

Ett av formålene med enhetstesting er å legge til rette for refaktorisering. Når man refaktoriserer endrer man kildekoden for en metode for å forbedre lesbarheten, vedlikeholdbarhet, struktur eller optimalisering [CC]. Dette gjøres uten å endre den eksterne oppførselen til metoden. Testene man har implementert brukes da for å sjekke at den eksterne funksjonaliteten forblir stabil.

7.1.2 Minneaksessering

Ved bruk av programmeringsspråk som C og C++ er det viktig å være nøye med hvordan man aksesserer minneområder. Det er fort gjort å skrive til uallokerte områder i disse språkene, glemme å frigjøre minne eller referere til null-pekere. Vår prototype er neppe fri for slike feil, men vi har forsøkt å luke ut det meste ved å bruke verktøy som kan detektere vanlige minneaksesseringsfeil.

7.2 Systemtest

Vi vil videre i dette kapittelet demontrere funksjonaliteten til prototypen og vise resultater fra ytelsestesting.

7.2.1 Testkjøring

Her følger prosedyre og skjermutskrift fra kjøring av systemet. Som beskrevet i vedlegg B, startes en kildestrøm fra VideoLAN slik:

```
$ vlc -L --sout udp:localhost:1235 mpegfil.mpg
```

Klient- og servermodul startes opp i hver sin konsoll:

```
$ ./vldrm
$ ./vldrmd
```

Og til slutt en mediaavspiller som kontakter klientmodulen:

```
$ vlc http://localhost:8888/MULTICAST/232.0.0.1
```

Klientmodul

Utskrift fra klientmodul. Gangen i programmet er som følger (tall i parantes refererer til linjenummer):

- Klientmodulen starter en HTTP-proxy på TCP port 8888 (3).
- Avspilleren kobler seg til og sender en forespørsel om å koble seg på multicast strøm på adressen 232.0.0.1 (4).
- Klientmodulen kobler seg til servermodulen via TLS og henter krypteringsnøkkelen (5).
- I løpet av avspillingen mottar klientmodulen en ny krypteringsnøkkel fem ganger. Hyppigheten her skyldes at programmet er kompilert med debugging og har da nøkkelskift hvert 20. sekund (8 – 17).
- Avspilleren kobler seg fra (19).
- Klientmodulen avsluttes (20 – 23).

```
1  vldrm[6459]: WARNING (4): DEBUG ENABLED
2  vldrm[6459]: INFO    (6): starting services
3  vldrm[6459]: INFO    (6): httpd server starting:
       0.0.0.0:8888
4  vldrm[6459]: DEBUG   (7): HTTPD: GET "/MULTICAST/232.0.0.1"
       HTTP/1.1
5  vldrm[6459]: DEBUG   (7): joining SSM multicast channel:
       129.241.102.55/232.0.0.1
6  vldrm[6459]: DEBUG   (7): libgcrypt 1.2.0 initialized
```

```
7  vldrm[6459]: NOTICE  (5): TLS: Handshake completed
8  vldrm[6459]: DEBUG   (7): TLS: new multicast key is
       available now.
9  vldrm[6459]: DEBUG   (7): USING KEY 3
10 vldrm[6459]: DEBUG   (7): TLS: new multicast key is
       available now.
11 vldrm[6459]: DEBUG   (7): USING KEY 4
12 vldrm[6459]: DEBUG   (7): TLS: new multicast key is
       available now.
13 vldrm[6459]: DEBUG   (7): USING KEY 5
14 vldrm[6459]: DEBUG   (7): TLS: new multicast key is
       available now.
15 vldrm[6459]: DEBUG   (7): USING KEY 6
16 vldrm[6459]: DEBUG   (7): TLS: new multicast key is
       available now.
17 vldrm[6459]: DEBUG   (7): USING KEY 7
18 vldrm[6459]: DEBUG   (7): TLS: closed
19 vldrm[6459]: DEBUG   (7): http session destroyed
20 vldrm[6459]: NOTICE  (5): caught signal SIGINT, terminating
21 vldrm[6459]: DEBUG   (7): TLS: close(), not connected
22 vldrm[6459]: DEBUG   (7): MediaClient terminated
23 vldrm[6459]: DEBUG   (7): httpd server destroyed
```

Servermodul

Utskrift fra servermodulen. Gangen i programmer er som følger (tall i parantes refererer til linjenummer):

- Konfigurasjonsfilen leses og gyldige verdier skrives ut (2 – 36).
- Multicast-strømmene starter opp (39 – 41). I dette eksemplet er det bare strømmen *ssm_multicast* som er aktiv. Den andre mottar ikke noe data den kan videresende.
- Det blir generert nye krypteringsnøkler (46 – 47, 56 – 70, 73 – 80).
- En klientmodul kobler seg til, autentiseres og får tilsendt en krypteringsnøkkel (48 – 55).
- Klientmodulen kobler seg fra (71 – 72).
- Servermodulen avsluttes (81 – 85).

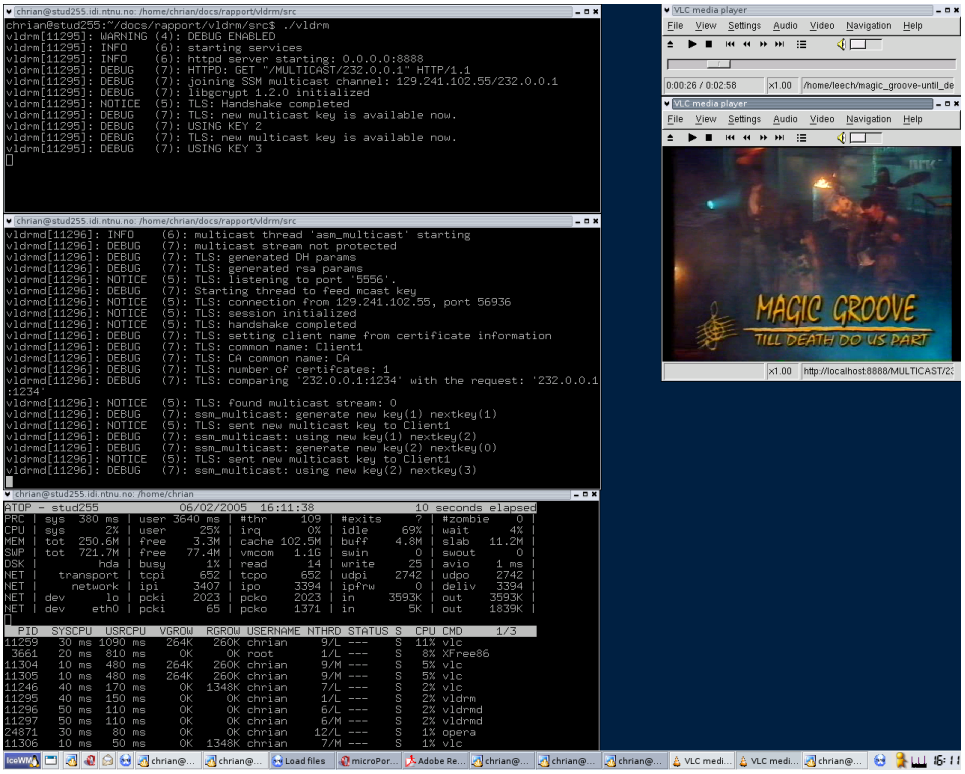
```
1  vldrmd[6453]: WARNING (4): DEBUG ENABLED
2  vldrmd[6453]: INFO    (6): reading config file: vldrmd.conf
3  vldrmd[6453]: DEBUG   (7): XML version="1.0" encoding="ISO
       -8859-1"
4  vldrmd[6453]: DEBUG   (7): config file version is 1.0
5  vldrmd[6453]: DEBUG   (7): conf: info="Test configuration."
6  vldrmd[6453]: DEBUG   (7): conf: daemon=0
7  vldrmd[6453]: DEBUG   (7): conf: debug=7 (valid=7)
8  vldrmd[6453]: DEBUG   (7): conf: TLS port=5556
9  vldrmd[6453]: DEBUG   (7): conf->status: enabled=0
10 vldrmd[6453]: DEBUG   (7): conf->status: ip="0.0.0.0"
```

```
11 vldrmd[6453]: DEBUG   (7): conf->status: port=8880
12 vldrmd[6453]: DEBUG   (7): conf->stream: id="ssm_multicast"
13 vldrmd[6453]: DEBUG   (7): conf->ssm_multicast: enabled=1
14 vldrmd[6453]: DEBUG   (7): conf->ssm_multicast: info="SSM
       multicast"
15 vldrmd[6453]: DEBUG   (7): conf->ssm_multicast: cipher="AES"
16 vldrmd[6453]: DEBUG   (7): conf->ssm_multicast: cipher mode
       ="CTR"
17 vldrmd[6453]: DEBUG   (7): conf->ssm_multicast->source: info
       ="UDP stream 1"
18 vldrmd[6453]: DEBUG   (7): conf->ssm_multicast->source: ip
       ="0.0.0.0"
19 vldrmd[6453]: DEBUG   (7): conf->ssm_multicast->source: port
       =1235
20 vldrmd[6453]: DEBUG   (7): conf->ssm_multicast->dest: info="
       SSM address"
21 vldrmd[6453]: DEBUG   (7): conf->ssm_multicast->dest: ip
       ="232.0.0.1"
22 vldrmd[6453]: DEBUG   (7): conf->ssm_multicast->dest: port
       =1234
23 vldrmd[6453]: DEBUG   (7): conf->ssm_multicast->dest: TTL=1
24 vldrmd[6453]: DEBUG   (7): conf->stream: id="asm_multicast"
25 vldrmd[6453]: DEBUG   (7): conf->asm_multicast: enabled=1
26 vldrmd[6453]: DEBUG   (7): conf->asm_multicast: info="ASM
       multicast"
27 vldrmd[6453]: DEBUG   (7): conf->asm_multicast: cipher="NONE
       "
28 vldrmd[6453]: DEBUG   (7): conf->asm_multicast: cipher mode
       ="NONE"
29 vldrmd[6453]: DEBUG   (7): conf->asm_multicast->source: info
       ="UDP stream 2"
30 vldrmd[6453]: DEBUG   (7): conf->asm_multicast->source: ip
       ="0.0.0.0"
31 vldrmd[6453]: DEBUG   (7): conf->asm_multicast->source: port
       =1236
32 vldrmd[6453]: DEBUG   (7): conf->asm_multicast->dest: info="
       ASM address"
33 vldrmd[6453]: DEBUG   (7): conf->asm_multicast->dest: ip
       ="239.0.0.1"
34 vldrmd[6453]: DEBUG   (7): conf->asm_multicast->dest: port
       =1234
35 vldrmd[6453]: DEBUG   (7): conf->asm_multicast->dest: TTL=1
36 vldrmd[6453]: NOTICE  (5): 2 stream(s) defined
37 vldrmd[6453]: INFO    (6): starting services
38 vldrmd[6453]: DEBUG   (7): libgcrypt 1.2.0 initialized
39 vldrmd[6453]: INFO    (6): multicast thread 'ssm_multicast'
       starting
40 vldrmd[6453]: INFO    (6): multicast thread 'asm_multicast'
       starting
41 vldrmd[6453]: DEBUG   (7): multicast stream not protected
42 vldrmd[6453]: DEBUG   (7): TLS: generated DH params
43 vldrmd[6453]: DEBUG   (7): TLS: generated rsa params
44 vldrmd[6453]: NOTICE  (5): TLS: listening to port '5556'.
45 vldrmd[6453]: DEBUG   (7): Starting thread to feed mcast key
46 vldrmd[6453]: DEBUG   (7): ssm_multicast: generate new key
       (1) nextkey(1)
47 vldrmd[6453]: DEBUG   (7): ssm_multicast: using new key(1)
       nextkey(2)
```

```
48 vldrmd[6453]: NOTICE  (5): TLS: connection from
       129.241.102.55, port 56315
49 vldrmd[6453]: NOTICE  (5): TLS: session initialized
50 vldrmd[6453]: NOTICE  (5): TLS: handshake completed
51 vldrmd[6453]: DEBUG   (7): TLS: setting client name from
       certificate information
52 vldrmd[6453]: DEBUG   (7): TLS: common name: Client1
53 vldrmd[6453]: DEBUG   (7): TLS: CA common name: CA
54 vldrmd[6453]: DEBUG   (7): TLS: number of certifcates: 1
55 vldrmd[6453]: NOTICE  (5): TLS: found multicast stream: 0
56 vldrmd[6453]: DEBUG   (7): ssm_multicast: generate new key
       (2) nextkey(0)
57 vldrmd[6453]: NOTICE  (5): TLS: sent new multicast key to
       Client1
58 vldrmd[6453]: DEBUG   (7): ssm_multicast: using new key(2)
       nextkey(3)
59 vldrmd[6453]: DEBUG   (7): ssm_multicast: generate new key
       (3) nextkey(0)
60 vldrmd[6453]: NOTICE  (5): TLS: sent new multicast key to
       Client1
61 vldrmd[6453]: DEBUG   (7): ssm_multicast: using new key(3)
       nextkey(4)
62 vldrmd[6453]: DEBUG   (7): ssm_multicast: generate new key
       (4) nextkey(0)
63 vldrmd[6453]: NOTICE  (5): TLS: sent new multicast key to
       Client1
64 vldrmd[6453]: DEBUG   (7): ssm_multicast: using new key(4)
       nextkey(5)
65 vldrmd[6453]: DEBUG   (7): ssm_multicast: generate new key
       (5) nextkey(0)
66 vldrmd[6453]: NOTICE  (5): TLS: sent new multicast key to
       Client1
67 vldrmd[6453]: DEBUG   (7): ssm_multicast: using new key(5)
       nextkey(6)
68 vldrmd[6453]: DEBUG   (7): ssm_multicast: generate new key
       (6) nextkey(0)
69 vldrmd[6453]: NOTICE  (5): TLS: sent new multicast key to
       Client1
70 vldrmd[6453]: DEBUG   (7): ssm_multicast: using new key(6)
       nextkey(7)
71 vldrmd[6453]: NOTICE  (5): TLS: client disconnected
72 vldrmd[6453]: DEBUG   (7): TLS: session destroyed
73 vldrmd[6453]: DEBUG   (7): ssm_multicast: generate new key
       (7) nextkey(0)
74 vldrmd[6453]: DEBUG   (7): ssm_multicast: using new key(7)
       nextkey(8)
75 vldrmd[6453]: DEBUG   (7): ssm_multicast: generate new key
       (8) nextkey(0)
76 vldrmd[6453]: DEBUG   (7): ssm_multicast: using new key(8)
       nextkey(9)
77 vldrmd[6453]: DEBUG   (7): ssm_multicast: generate new key
       (9) nextkey(0)
78 vldrmd[6453]: DEBUG   (7): ssm_multicast: using new key(9)
       nextkey(10)
79 vldrmd[6453]: DEBUG   (7): ssm_multicast: generate new key
       (10) nextkey(0)
80 vldrmd[6453]: DEBUG   (7): ssm_multicast: using new key(10)
       nextkey(11)
81 vldrmd[6453]: NOTICE  (5): caught signal SIGINT, terminating
```

```
82 vldrmd[6453]: INFO     (6): shutting down services
83 vldrmd[6453]: DEBUG    (7): Destructing object in
       TlsMcastKeyChanger
84 vldrmd[6453]: DEBUG    (7): multicast thread 'ssm_multicast'
       destroyed
85 vldrmd[6453]: DEBUG    (7): multicast thread 'asm_multicast'
       destroyed
```

Figur 7.1 viser alle komponentene kjørende på en maskin. De tre konsollvinduene til venstre viser ovenifra og ned: utskrift fra klientmodulen, servermodulen og en systemmonitor. Øverst til høyre er streaming-serveren og videoavspilleren under denne.



Figur 7.1: Skjermbilde med hele systemet kjørende på en maskin

7.2.2 Manglende eller ugyldig sertifikat

I tilfellet der klientmodulen har et ugyldig sertifikat vil denne bli avvist når det kobler seg til servermodulen. De neste linjene er et utdrag fra loggen til servermodulen som viser en klient som kobler seg til og blir avvist.

```
1 vldrmd[30775]: NOTICE   (5): TLS: connection from
      129.241.102.55, port 57556
2 vldrmd[30775]: NOTICE   (5): TLS: session initialized
3 vldrmd[30775]: NOTICE   (5): TLS: handshake completed
```

```
4  vldrmd[30775]: WARNING (4): TLS: The certificate has been
       revoked.
5  vldrmd[30775]: ERROR   (3): TLS: No certificate was found!
6  vldrmd[30775]: DEBUG   (7): TLS: session destroyed
```

Linjene nedenfor er utskrift av loggen til klientmodulen som foretar oppkoblingen mot servermodulen som vist i linjene over.

```
1  vldrm[30850]: WARNING (4): DEBUG ENABLED
2  vldrm[30850]: INFO    (6): starting services
3  vldrm[30850]: INFO    (6): httpd server starting:
       0.0.0.0:8888
4  vldrm[30850]: DEBUG   (7): HTTPD: GET "/MULTICAST/232.0.0.1"
        HTTP/1.0
5  vldrm[30850]: DEBUG   (7): joining SSM multicast channel:
6  129.241.102.55/232.0.0.1
7  vldrm[30850]: DEBUG   (7): libgcrypt 1.2.0 initialized
8  vldrm[30850]: NOTICE  (5): TLS: Handshake completed
9  vldrm[30850]: ERROR   (3): TLS: Error in reciving key on
       client: A TLS packet
10 with unexpected length was received.
11 vldrm[30850]: DEBUG   (7): caught signal SIGPIPE, ignoring
12 vldrm[30850]: DEBUG   (7): TLS: closed
13 vldrm[30850]: WARNING (4): TLS: Not able to shut down
       connection
14 vldrm[30850]: ERROR   (3): TLS: Unable to get key, start
       method failed
15 vldrm[30850]: ERROR   (3): Access denied to MULTICAST
       /232.0.0.1:1234
16 vldrm[30850]: DEBUG   (7): http session destroyed
17 vldrm[30850]: NOTICE  (5): caught signal SIGINT, terminating
18 vldrm[30850]: DEBUG   (7): TLS: close(), not connected
19 vldrm[30850]: DEBUG   (7): MediaClient terminated
20 vldrm[30850]: DEBUG   (7): httpd server destroyed
```

Når klientmodulen blir avvist ved tilkobling til servermodulen vil den returnere en HTTP-feilmelding tilbake til klienten. Slik blir det opp til avspilleren å informere brukeren av tilgangen ble avvist. Denne er formatert som følger:

```
1  HTTP/1.1 403 Forbidden
2  Server: httpd
3  Connection: close
4  Accept-Ranges: bytes
5  Last-Modified: Mon, 06 Jun 2005 09:23:42 GMT
6  Date: Mon, 06 Jun 2005 09:23:42 GMT
7  Content-Length: 246
8  Content-Type: text/html; charset=iso-8895-1
9
10 <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
11 <html><head>
12 <title>403 Forbidden</title>
13 </head><body>
14 <h1>403 Forbidden</h1>
15 <p>Access denied</p>
16 <hr>
17 <address>httpd Server at stud255.idi.ntnu.no Port 8888</
       address>
18 </body></html>
```

7.2.3 Oppstartstid

Oppstartstid av en videostrøm ble målt til 0,2 sekunder. Dette er tidsbruk fra avspilleren kontakter klientmodulen og til avspilleren igjen får levert første dekrypterte pakke. Oppkobling av en sikker kanal er også inkludert i denne tiden. Oppstartstiden er beskrevet i avsnitt 3.2 til å ikke overstige ett sekund. Denne tiden skal også inkludere komponentene utenfor vårt system, samt forhåndbuffring av data i avspilleren. Det er derfor fortsatt rom for å redusere oppstartshastigheten i prototypen. Dette kan for eksempel gjøre ved å holde den sikre kanalen åpen også når det ikke er noen aktive strømmer. Slik vil oppkoblingstiden til forbindelsen kunne trekkes fra oppstartstiden.

7.2.4 Ressursberegning

Kryptering er en krevende operasjon, og dette setter en grense for hvor mange datastrømmer en enkelt datamaskin kan kryptere av gangen. På klientmodulen vil det til en hver tid ikke være behov for å dekryptere mer enn en datastrøm av gangen. Det er derfor bare kritisk at maskinvaren i denne enden av systemet klarer å håndtere dette.

På servermodulen er derimot kravene høyere. For multicast-strømmer må servermodulen ha nok prosessorkraft til å kryptere så mange datastrømmer som det er kanaler som skal distribueres. Antall brukere spiller liten rolle. Det er klart at i vår løsning er det også en begrensing i hvor mange klienter som kan opprette en sikker unicast-kanal for overføring av nøkler, men skaleringen av disse tilkoblingene vil ikke være like kritisk. Unicast-krytering vil klart ha en utfordring ved å kunne kryptere en datastrøm per bruker.

I første omgang er denne prototypen tenkt brukt i for eksemel en IPTV-installasjon i et boretslag. Her vil brukerbasen være i størrelsesorden et par hundre. Lengre frem i tid kan det være interessant å skalere et slikt system til for eksempel hoteller eller DSL-kunder. Her vil antall brukere kunne være flere tusen.

Forutsetninger

Testene og målingene videre i dette avsnittet er basert på en del ideelle omstendigheter og forutsetninger:

- Testene er gjennomført på vanlige arbeidsstasjoner med relativt lite prosessorkraft eller spesiell maskinvare.
- Tidsmålingene av krypteringen er foretatt isolert sett og ser bort fra hvilke datamenger maskinen for eksempel klarer å dytte ut gjennom nettverkskortet.

Tabell 7.1 og 7.2 viser resultatet fra målinger foretatt på servermodulen. Målingene er et gjennomsnitt av ca. 10 sekunders kjøring. Dette er sammenlignbart med målingene av chifferalgoritmer i vedlegg C.3. I tillegg til naiv

kryptering, som er hovedmetoden for kryptering i prototypen, inkluderer testen selektiv kryptering av typen som er beskrevet i avsnitt 8.4.1. Figur 7.2 og 7.3 viser en grafisk fremstilling av disse tabellene.

Videostrømmen hadde en datarate på 5896 kb/s (det vil si 754 688 bytes per sekund). Dette er en datarate som gir video med høy kvalitet. Det som er interessant å se er hvor mange megabytes per sekund det er multig å kryptere. Det er også grovt beregnet hvor mange datastrømmer denne maskinen ideelt vil klare å kryptere samtidig (kolonne "Str." i tabell 7.1 og 7.2).

Metode	Modus	MB/s	Gj.snitt (μs)	Median	Min	Max	Str.
Naiv	CTR	6,71	187	180	166	1406	9
Selektiv	CTR	15,31	82	79	64	670	21
Naiv	CFB	11,62	108	108	86	552	16
Selektiv	CFB	21,64	58	51	45	425	30

Tabell 7.1: Benchmark av krypteringshastighet, Pentium 3 1GHz

Metode	Modus	MB/s	Gj.snitt (μs)	Median	Min	Max	Str.
Naiv	CTR	10,91	115	112	108	978	15
Selektiv	CTR	26,15	48	45	43	428	36
Naiv	CFB	16,09	78	75	69	800	22
Selektiv	CFB	32,18	39	37	34	350	45

Tabell 7.2: Benchmark av krypteringshastighet, Pentium 4 1,8GHz

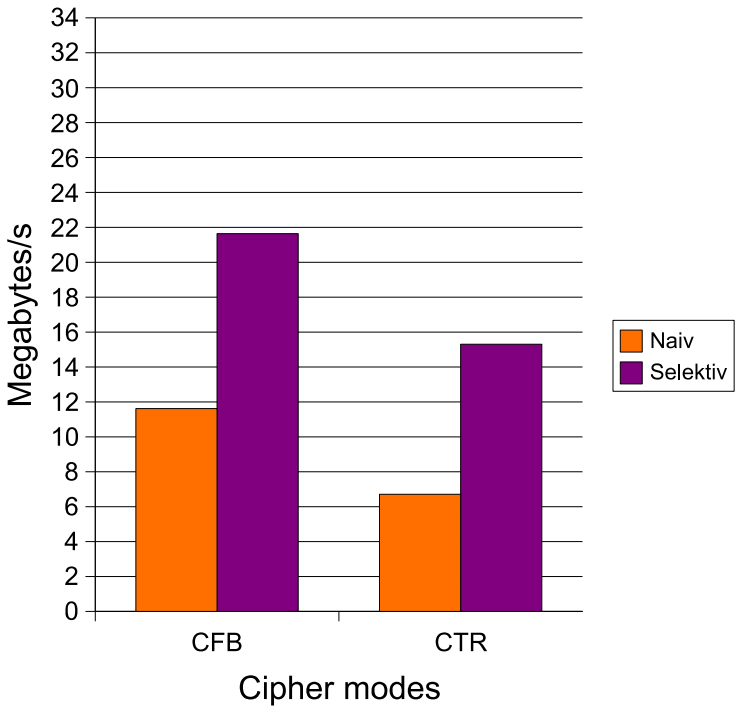
Ligning 7.1 viser en grov beregning av hvor mange datastrømmer som kan krypteres samtidig av systemet. I praksis er dette veldig optimistisk. Tester viste at ved det første tilfellet i tabell 7.1, som er beregnet til å håndtere 9 datastrømmer, vil denne maskinen ikke klare å kryptere mer enn 7 datastrømmer med den samme dataraten. Ved 9 datastrømmer brukte maskinen 99% prosessorkraft. Ved 7 datastrømmer sankt prosessorbruken ned til mellom 85% og 92%.

$$\frac{1}{ppp * \frac{r}{bpp}} \approx s \tag{7.1}$$

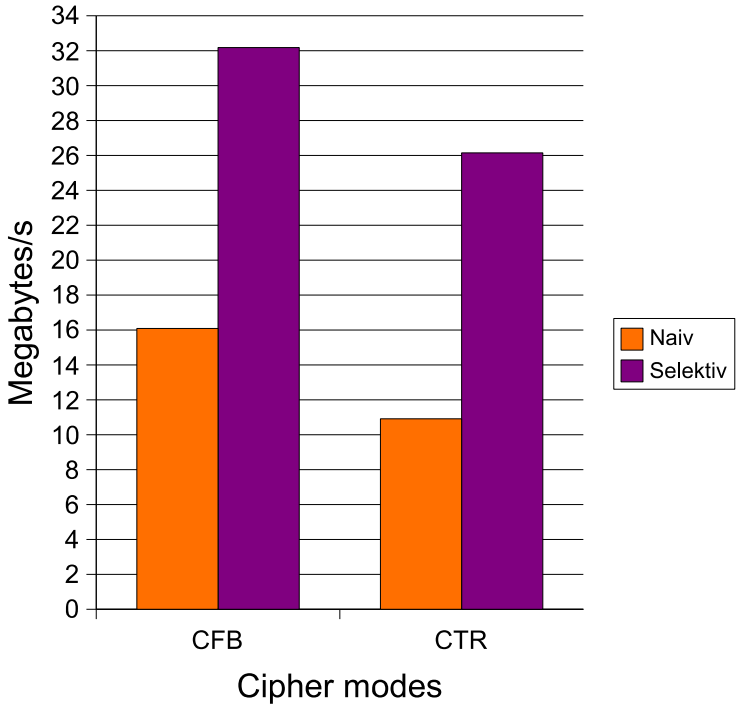
ppp	prosesseringstid per datapakke
r	datarate (bytes per sekund)
bpp	antall bytes per pakke (se avsnitt 6.2)
s	antall datastrømmer

Tabell 7.3: Symbolforklaring for ligning 7.1

$$\frac{1}{0,000187 * \frac{754688}{1316}} \approx 9 \tag{7.2}$$



Figur 7.2: Benchmark av vldrmd, Pentium 3 1GHz



Figur 7.3: Benchmark av vldrmd, Pentium 4 2,4GHz

7.2.5 Eksekveringsprofil

Gprof er en applikasjon for å produsere en eksekveringsprofil for et C/C++ program. Denne profilen kan brukes til å finne ut av hvor mange ganger funksjonene i programmet blir eksekvert og hvor lang tid disse bruker. Programmet som skal profileres må være kompilert med et spesielt valg (`-pg` i gcc-kompilatoren). Programmet kjøres så normalt og informasjonen lagres i filen `gmot.out`.

Et problem med *gprof* er at for flertrådsprogrammer (som servermodulen) vil den bare profilere hovedtråden. For å løse dette kan programbiblioteket `gprof-helper.c` kompileres og lastes sammen med flertrådsprogrammet. Alle trådene vil da bli eksekvert. Se `gprof-helper.c` i vedlegg G for hvordan dette fungerer.

Tabell 7.4 viser deler av den flate eksekveringsprofilen for servermodulen. Manualsiden til *gprof* (`man gprof`) gir en grundigere forklaring på de forskjellige kolonnene. Men utfra tabellen kan man få et inntrykk av hvilke funksjoner det kan lønne seg å optimalisere, for eksempel ved å se på antall kall sammen med tiden brukt av funksjonen.

Ulempen med denne tabellen slik den er nå er at spesielt krypteringsfunksjonen, som vi vet er en krevende operasjon, ikke blir medberegnet. Dette er fordi funksjonen `Cipher::encrypt()` bare kaller en funksjon i *gcrypt*-biblioteket. Dette biblioteket måtte også vært kompilert med profileringsinformasjon for å bli medberegnet.

Fra eksekveringsprofilen er det også mulig å generere en funksjonskallgraf. Vedlegg F inkluderer en slik graf for server- og klientmodulen. Her er det mulig å ane noe av strukturen i programmet og kan for eksempel brukes til å få en oversikt over sentrale kontrollfunksjoner og hvor forskjellige funksjoner blir kalt fra.

7.2.6 Enhetstester

Det er implementert et sett med enhetstester for et sett med viktige klasser. Disse kan kompileres og kjøres med kommandoen `make check`. Resultatet av kjøring av enhetstestene er vist nedenfor. Hvert punktum representerer en vellykket test.

```
1  Cipher CTR test: ...
2  Cipher test: ........
3  CipherKey test: .......
4  Config class: .....
5  ConfigServer class: ...
6  ConfigStream class: .......
7  ConfigTarget class: .
8  HttpRequest test: ...
9  Mpi test: .......
10 UdpReader/UdpWriter communication: .
11
12 tests summary: ok:45
```

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
21.43	0.03	0.03				MulticastServer::run()
14.29	0.05	0.02				Mpi::operator+(unsigned int)
7.14	0.06	0.01	19445	0.00	0.00	CipherKey::getNoncelen() const
7.14	0.07	0.01	6481	0.00	0.00	Mpi::get(unsigned char*, unsigned int, bool)
7.14	0.08	0.01	6481	0.00	0.00	Cipher::activateIV()
7.14	0.09	0.01	6481	0.00	0.00	Cipher::setIV(unsigned char const* , int)
7.14	0.10	0.01	2	5.00	7.50	ConfigServer::processStreamNodes(xmlNode*, ConfigStream*)
7.14	0.11	0.01				Mpi::operator¿(unsigned int) const
7.14	0.12	0.01				Mpi::operator¡(Mpi const&) const
7.14	0.13	0.01				Cipher::isIVSet() const
3.57	0.14	0.01	2	2.50	2.50	Cipher::mapName(char const*)
3.57	0.14	0.01	2	2.50	2.50	Cipher:: Cipher()
0.00	0.14	0.00	12962	0.00	0.00	Mpi::operator+=(unsigned int)
0.00	0.14	0.00	12962	0.00	0.00	Mpi::operator++(int)
0.00	0.14	0.00	6483	0.00	0.00	UdpReader::read(unsigned char*, int)
0.00	0.14	0.00	6481	0.00	0.00	Cipher::encrypt(unsigned char*, unsigned int, unsigned char const*, unsigned int)
0.00	0.14	0.00	6481	0.00	0.00	UdpWriter::write(void const*, unsigned int)

Tabell 7.4: Deler av eksekveringsprofilen for servermodulen

```
13 PASS: testall
14 ==================
15 All 1 tests passed
16 ==================
```

7.2.7 Kjente feil

Dette er som kjent et prototypesystem, og alle feil er ikke luket ut av systemet. Under testing ble følgende alvorlige feil oppdaget:

- Ved ytelsestestene med høy prosessorbruk (99%) avsluttes servermodulen med en *abort*-filmelding fra *gcrypt*. Kan være en feil ved flertrådsbehandlingen i *gcrypt*, eller at systemet ikke klarer å allokere nok ressurser under slik prosessorlast.
- Sporadiske *segmentfeil* (minnefeil) ved avslutning av servermodulen. Feilen er bare delvis reproduserbar og skyldes mest sannsynlig en *race condition* ved avslutningen av multicast-trådene.
- En alvorlig minnelekkasje ved nedkobling av TLS-forbindelsen. Vanskelig å finne kilden, men kan være mangel på de-allokering av TLS-ressurser.
- Noen mindre minnelekkasjer som rapporteres av Valgrind, men det går ikke klart frem hva som forårsaker lekkasjene.

Del IV

Avslutning

Kapittel 8

Diskusjon

Kapittelet beskriver hvilke valg vi har har stått ovenfor og løsningene vi kom frem til. Vi kommer også med forslag om hvordan vi kan beskytte IPTV og diskuterer våre erfaringer og løsninger knyttet til vår implementering.

8.1 Trusselanalyse

Som vi drøftet i [AM04] vil DRM-beskyttelse alltid ha svakheter og være teoretisk mulig å bryte fordi det utleveres både *lås og nøkkel* til sluttbrukeren (i praksis en kryptografisk beskyttelse og en nøkkel til denne). Dette gir mange potensielle måter for brukere å bryte DRM-beskyttelsen på (se 8.3.1).

DRM tas i bruk fordi rettighetsinnehavere krever en form for tilgangsbeskyttelse. Det viktigste for rettighetsinnehaverene er kravet om kryptering av overføringen av innholdet. Dette vil hindre andre enn godkjente brukere å ta i få tilgang til innholdet, og vanskeliggjøre avlytting og tapping av forbindelsen.

Scenariet er ofte at distribuert video over Internett enten er ansett som verdiløs og dermed ikke kryptert, eller så tas det i bruk krevende krypteringsmetoder [GMDS98]. Beskyttelse mot hver enkelt trussel vil være en avveining mellom kostnader og verdien til innholdet. Det er mulig å distribuere video i lukkede nett, påført vannmerker på datastrømmen og eget forseglet avspillerutstyr. Dette ville ha hevet både graden av beskyttelse og prisen på realiseringen, men det vil ikke være hensiktsmessig, fordi vi må anta at få brukere er villige til å betale for denne graden av beskyttelse og teknologikostnadene det kan føre med seg. I mange IPTV-systemer kan det være tilstrekkelig beskyttelse å forringe videostrømmen nok til at den ikke egner seg for visning. Det er et spørsmål om hvor høy verdi strømmen har, og hvor *kostbart* det skal være å reversere beskyttelsen. I tillegg til spørsmålet om grad av beskyttelse, må hver IPTV-leverandør stille spørsmålet om hvilke deler i systemet som skal beskyttes. Det kan også være aktuelt å beskytte forespørselen fra klienten. Dette kan være aktuelt der det er sensitiv infor-

masjon, som for eksempel hvilke børskurser en aksjemegler interesserer seg for. Eller er vi villige til å la verden (eller naboen) overvåke hva vi ser på TV? Trenger man sikkerhet for at ingen utsetter deg for feilinformasjon ved å overta sendingen? Integritetssjekk kan være løsningen på sistnevnte problem.

8.2 Evaluering av prototypen

Det er viktig å merke seg at løsningen baserer seg på aksesskontroll og ikke kopikontroll. For at vår løsning skal bli et fullverdig DRM-system er det flere viktige elementer som i tillegg må på plass. Løsningen tar ikke hensyn til opptak av video lokalt hos klienten. Dette går tilbake til hvilke problemer dette systemet er ment å løse. Vår IPTV-løsning baserer seg på en "forlengelse av videokabelen", det vil si at opptaket gjøres sentralt på en server og brukerne ikke skal ha mulighet til å lagre dette lokalt. Hvis det er fare for at brukeren likevel kobler til utstyr for i opptak, bør videostrømmen vannmerkes, for å kunne spore videre distribusjon. Vannmerkingsprosessen vil kreve for mye prosesseringsressurser i vår transparente løsning. En eventuell vannmerking bør utføres av enheten som koder strømmen før den blir kryptert av vårt system. Hvis vi skulle stått for vannmerking ville vi måtte dekode og så kode strømmen igjen, som ville ført til mye dobbeltarbeid. Dette er egentlig samme problemstilling vi står ovenfor med flere metoder for selektiv kryptering (se avsnitt 2.7.3 og 8.4.1).

8.2.1 Brukervennlighet

En vanlig bruker av IPTV-systemet skal i utgangspunktet ikke trenge å vite at det eksisterer noen form for beskyttelsesmekanisme. Det som bør forbedres ved prototypen er tilbakemeldinger ved feilsituasjoner. Det mest opplagte er at brukeren bør få en informativ tilbakemelding hvis man prøver å aksessere noe man ikke har tilgang til. IPTV-applikasjonen må da kunne tolke feilmeldingen og for eksempel gi brukeren mulighet til å kjøpe seg tilgang.

8.2.2 Kompleksitet

En-trådet klientmodul

Det er spesielt ett valg vi gjorde i utviklingen som fikk følger vi ikke så på forhånd. Dette var da vi bestemte oss for å gjøre avspilleren en-trådet. Valget ble gjort for å forenkle klienten, både for en enkel programflyt og for å gjøre klienten *lett* i forhold til hvilke ressurser potensielle set-top-bokser har til rådighet.

Det vi oppdaget var at mediaavspillere som bruker flere tråder, aksesserer HTTP-grensesnittet til klienten fra mer enn en tråd av gangen. Disse avspillerne vil derfor ikke fungere med applikasjonen slik den er nå. Eksempler på avspillere vi har funnet til *ikke* å fungere er *mplayer* og Windows Media Player. Avspillere som fungerer uten problemer er *VideoLAN* og *Xine*.

8.2.3 Er kravspesifikasjonen oppfylt?

Vår prototype har som hovedfunksjon å hjelpe oss med å besvare spørsmålene vi har stilt i problemstillingen. Det er også et håp om at det vi har utviklet kan være til nytte i scenarioet som er utgangspunktet for problemstillingene, beskyttelse av IPTV-sendinger.

Kravspesifikasjonen er satt opp som et dokument vi har kunnet følge for å finne ut av hvilken funksjonalitet et slikt system *bør* ha. I vår situasjon har vi vært nødt til å prioritere enkelte punkter av den som er viktigst for våre faglige problemstillinger. Alle punkter i kravspesifikasjonen er derfor ikke med i prototypen slik den er ved innlevering av denne rapporten. Det har vært både tidsmessige og behovsmessige grunner til dette.

I resten av dette avsnittet vil vi gå igjennom noen viktige avvik fra kravspesifikasjonen.

Tilgangskontroll

Det viktigste avviket mellom kravspesifikasjonen og implementasjonen er mangel på funksjonalitet for tilgangskontroll for spesifikt innhold (avsnitt 3.3.3, punkt 2). Dette er en absolutt nødvendighet for at systemet skal være nyttig i praksis. Punktet ble nedprioritert av tidsmessige årsaker. Slik systemet fungerer nå vil man ha tilgang på alt innhold så lenge man har et sertifikat som autentiserer brukeren.

Tanken er at dette må integreres mot eksterne løsninger som setter begrensninger for bruk i forhold til hva som er betalt for. For eksempel eksterne betalings- og administrasjonssystemer.

Unicast-kryptering

En del av funksjonaliteten vi ikke prioriterte i implementeringen var kryptering av unicast-strømmer (3.3.1, 1a). Det viktigste for oss var å kunne se hvordan multicast-krypteringen ville fungere. Unicast-kryptering er riktignok det som vil være den største utfordringen for skalerbarheten til systemet, da hver forbindelse fra klient til server vil kryptere en datastrøm. I motsetning til multicast, hvor serveren krypterer bare en strøm for en eller flere klienter samtidig.

Det som mangler for å implementere denne funksjonaliteten er å få klientmodulen til å opprette en TCP-forbindelse for dataoverføring fra servermodulen, samt å implementere en HTTP-proxy på servermodulen.

Mellomlager for krypteringsnøkler

Ved utarbeidelsen av kravspesifikasjonen så vi for oss at klientmodulen kunne ha nytte av et lokalt mellomlager for nøkler (3.3.1, 2d). Slik implementasjonen utviklet seg ble ikke dette nødvendig. Det eneste form for nøkkellager applikasjonen har er for å lagre gjeldende og neste nøkkel. Slik har klientmodulen klar neste nøkkel når servermodulen signaliserer et nøkkelbytte. Dette gjelder kun innenfor en sesjon, og ved bytte av strøm hentes nye krypteringsnøkler ned fra servermodulen.

Tilleggsfunksjonalitet

Funksjonalitet som ikke er spesifisert i kravspesifikasjonen er også kommet frem under utviklingen. Dette er i stor grad funksjonalitet som har vært til hjelp underveis, eller generell kodegjenbruk. Det er også naturlig å oppdage ukjente praktiske problemstillinger med implementeringen som ikke var tenkt på fra start.

- *Enkel HTTP statusserver.* Da vi allerede hadde utviklet HTTP-serverfunksjonalitet for den klientbaserte HTTP-proxyen, var det enkelt å gjenbruke denne koden som basis for en slik statusserver på servermodulen. Denne er ikke ferdigstilt. Tanken er at man her kan se status over de forskjellige multicast-strømmene og oppkoblede klienter. Mer avansert funksjonalitet kan være et administrativt grensesnitt for å endre innstillinger på servermodulen mens den kjører.

- *SQL-basert lisenslagring.* Det er påbegynt et grensesnitt for å hente lisens- og tilgangsinformasjon fra en SQL-server. Slik funksjonalitet vil gjøre vår løsning enklere å integrere med eksisterende betalingsløsninger.

- *Prioritetsbasert logging.* Det er implementert et loggsystem som gjør feilsøking av systemet enklere. Dette kan settes opp til å logge for eksempel til en fil, skjerm eller systemloggen. Nye klasser kan enkelt implementeres slik at man for eksempel kan logge direkte til en SQL-server. Hvor mye informasjon programmet skriver ut under kjøring kan konfigureres. Høyeste nivå er *debug* (nivå 7, alle meldinger skrives ut, kan være ganske detaljert og mye informasjon) og laveste er *emergency* (nivå 0, kun systemkritiske meldinger).

- *Fleksibelt utskifte av krypteringsmetode.* Ved å bruke programbiblioteket gcrypt, som implementerer flere forskjellige krypteringsalgoritmer, har systemet mulighet til å endre algoritme hvis man skulle finne ut at en annen algoritme vil være bedre egnet.

8.3 Beskyttelse

Utfordringene ligger i distribusjon av nøkler, dynamisk nøkkelbytte, valg av riktig algoritme og sikkerheten i implementasjonen. Metodene og algoritmene må implementeres på en korrekt måte for å kunne garantere en viss grad av sikkerhet.

Vi har valgt en struktur med en separat kontrollkanal til distribusjon av nøkler og kontroll av identitet. Alternativet hadde vært å distribuere nøklene i samme datastrøm som mediainnholdet. Hvis neste krypteringsnøkkel leveres kryptert i strømmen vil du også ha vanskeligheter med å luke ut klientene som ikke har betalt. Siden kontrollkanalen kan brukes til levering av både multicast- og unicast-nøkler fant vi dette mest hensiktsmessig med en kontrollkanal. Synkroniseringen ved nøkkelbytte signaliseres i headeren på den krypterte strømmen.

Vi har eksperimentert med lette krypteringsalgoritmer (se avsnitt 8.4.2), og ikke funnet noen tilfredsstillende muligheter. Selektiv kryptering (se avsnitt 8.4.1) er neste skritt i denne retningen, som også gir tilfredsstillende beskyttelse av videostrømmen. Som et første steg er det brukt naiv kryptering i vår implementasjon.

8.3.1 Sikkerhet

Ulovlig tilgang til innholdet i IPTV kan oppnås flere steder i systemet. Det beskyttede innholdet eksisterer i sin helhet: på server, under overføring til klient og under avspilling hos bruker. Vi tar kort for oss hva vi ser på som de største sikkerhetsrisikoene for IPTV sett fra vårt system, og hva som kan gjøres for å heve sikkerhetsnivået:

- Opptak og lagring av dekryptert strøm på kompromittert klient. Hvis IPTV-løsningen skal være brukbar, må den ha mulighet til å vise frem ukrypterte digitale TV-signaler til brukeren. Disse kan i teorien alltid kopieres, med mindre man har et *trusted computing* system [Wikb], der serveren kan stole på klienten. Vannmerker lagt til i strømmen kan benyttes for å kunne spore opp den kompromitterte klienten hvis innholdet blir distribuert videre.

- Speiling/kopiering av en set-top-boks med gyldig sertifikat. Dette kan hindres ved at et sertifikat på hver set-top-boks kun kan brukes fra en IP-adresse om gangen. En slik løsning kan føre til mistenkeliggjøring av brukere som er uvitende om at deres boks har blitt speilet. Hvis vi ikke identifiserer brukeren basert på IP-adresse eller maskinvare-ID, får vi et slikt kopieringsproblem. Man kan kontrollere at for eksempel ikke flere enn 2 bruker-sertifikater er i bruk om gangen, men dette kan fort mistenkeliggjøre brukere ved feil. Hvis vi baserer identifiseringen på nettverksadressen eller en unik maskinvare-ID, vil vi fremdeles kunne

flytte over sertifikatet til en ny set-top-boks. Vi bør i dette tilfelle kontrollere klientmaskinvaren hver gang, som igjen krever at serveren kan stole på at klienten ikke blir endret.

- Nytteverdien ved avlytting av overføringen mellom server og klient kan til en viss grad hindres av kryptering. Strømmen kan tas opp av tyvlytteren og deretter dekrypteres, med nok datakraft og tid vil det være teoretisk mulig å knekke krypteringen. Hovedmålet her må være at det ikke skal lønne seg framfor å faktisk kjøpe seg tilgang.

- Paritetskontroll på meldingene bør utføres. Uten dette vil en tredjepart kunne introdusere meldinger inn i systemet. Headeren er ikke kryptert og tredjeparten kan lese nøkkel-ID og meldings-ID og sende meldinger til systemet som kan føre til at krypteringen kommer ut av sync (for eksempel ved å endre nøkkel-ID). En integritetssjekk på hele meldingen (inkludert header) ville også kunne hindre dette. Derimot vil dette føre til en økning av meldingsstørrelsen.

8.4 Ressursbruk og ytelse

Noe av det største problemet med beskyttelse av IPTV er høy ressursbruk ved sanntids-kryptering av videostrømmen. En rask prosessor har ikke noen problemer med å kryptere flere media-strømmer samtidig ved anvendelse av AES på alle data (naiv kryptering). Spørsmålet er hvor mange. For vårt system vil det til en hver tid være et statisk antall multicast-strømmer, og dette er noe man kan ta høyde for ved dimensjonering av systemet. Se avsnitt 7.2.4.

Det er klart at en programvarebasert løsning som her er presentert, ikke nødvendigvis er ideelt. For optimal ytelse og vil et dedikert maskinvaresystem være best. Grunnen vil valg av en programvarebasert løsning er behovet for å kunne bruke billig hyllevare til å sette opp en tjeneste med vanlige servermaskiner.

8.4.1 Selektiv kryptering

Ved et høyt antall multicast-strømmer, og spesielt ved kryptering av unicast-strømmer til hver klient, vil naiv kryptering med AES få skaleringsproblemer. Et alternativ kan være å se på metoder for selektiv kryptering. Altså at bare deler av datastrømmen blir kryptert. Vårt problem med noen av metodene for selektiv kryptering er at video-strømmen allerede er komprimert. Flere av metodene må i så fall systemet dekode og så kode video-strømmene på nytt, noe som sannsynligvis ville ført til at ressursene brukt på dekoding/koding ville gått opp i det som var spart med mindre data å kryptere. Dette er et resultat av vår problemstilling for en transparent løsning.

Hvis vi ser på de forskjellige metodene nevnt i avsnitt 2.7.3 er *one-time-pad*-metoden et alternativ som kan redusere mengden data som krypteres, uten at man nødvendigvis trenger å vite så mye om innholdet. Vi kan anta at video-strømmene som skal krypteres av vårt system er i MPEG TS-format. Det vil si at hver pakke har en størrelse på 188 bytes, inkludert en header. Ved bare å kryptere TS-headeren vil det gjøre det veldig vanskelig å demultiplekse TS-pakkene igjen. Men som nevnt i avsnitt 2.7.3 kan header-informasjon ofte rekonstrueres.

Et alternativ er å kryptere for eksempel en fjerdedel av TS-pakken (47 bytes), noe som også vil kryptere TS-headeren på 4 bytes. Som nevnt i avsnitt 2.5.2 inneholder denne headeren blant annet feltet PID, slik at ved å kryptere denne vil man ikke vite hvordan man skal demultiplekse datastrømmen. Resten av dataene vil XORes sammen med dette 47 bytes kryptogrammet. Denne bruken av XOR vil ha en snev av *security through obscurity*[1] ved seg, fordi man bare kan ta de 47 første bytene og XORe med resten for å få klarteksten tilbake igjen. Dette forutsetter at metoden er kjent. Sikkerheten vil fortsatt ligge i den krypterte delen av TS-pakken.

For at klientmodulen skal vite hvilken metode som er brukt av server-modulen for å kryptere datastrømmen må informasjon om dette overføres i begynnelsen av sesjonen. Ved en utvidelse av nøkkelutvekslingsformatet kan dette overføres sammen med resten av parameterne. Ekstra parametere som trengs er størrelsen på pakkene (alternativt kan denne defineres til å være 188 bytes) og hvor mye av dataene i pakken som skal krypteres. Altså to heltallsverdier.

8.4.2 *Lett* kryptering

Vi gjorde forsøk med forskjellige krypteringer, men fant ingen som var akseptabelt gode eller raskere enn AES (se avsnitt C.3). Det er mulig å utvikle enklere former for scrambling, men for de som har erfaring med metoder for scrambling av TV-signaler i for eksempel satellitt-TV, blir dette mer *security trought obscurity* enn en akseptabel beskyttelse. Denne formen for scrambling vil gi en enkel forringelse av videobilde. Dette vil vi ikke anbefale, selv om det ved multicast-kryptering blir brukt hyppige nøkkelskift.

8.4.3 Lastbalansering

For multicast-strømmer vil en kraftig maskin i stor grad kunne håndtere de antall datastrømmer som er ønskelig. Men for unicast VoD vil én maskin bare kunne håndtere et begrenset antall tilkoblede klienter. Dette kan løses ved

[1] *Security through obscurity* er et begrep brukt om beskyttelsesmekanismer som baserer seg på at selve mekanismen holdes hemmelig. I motsetning til moderne kryptografi som baserer sikkerheten på at krypteringsnøkkelen holdes hemmelig.

lastbalansering av systemet, det vil si å fordele oppgavene på flere maskiner. Det er flere metoder for å innføre lastbalansering i løsningen:

- *Round-robin DNS*. Dette baserer seg på at det settes opp en DNS-server (dynamic name service) som kan svare med et sett av IP-adresser til forskjellige maskiner som kjører en versjon av servermodulen. Dette vil ikke kreve spesielle endringer i systemet.

- *Multiprosessor*. Prototypens bruk av separate tråder for krypteringsoperasjonene vil kunne dra nytte av et multiprosessorsystem slik at disse prosessene kan fordeles på tilgjengelige prosessorer.

- *Distribuert system*. En løsning som krever betydelige endringer i systemet. Et forslag vil være å separere ut systemet for nøkkelbehandling og synkronisering i en sentral enhet. Denne kan så styre forespørsler fra klientmodulene til et distribuert sett med servermoduler.

8.5 Sen implementering

Vi skulle arbeide for å støtte et eksisterende system som hele tiden er i utvikling. Noe av det viktigste ved prototypen vår er at den ikke er avhengig av noen av ende-systemene den skal beskytte. Den fungerer som en sikker tunnel for media-strømmen fra server til klient. Dette gir fleksibilitet på både server- og klientsiden. Derimot må beskyttelsesmekanismen integreres med betalingsystemet/abonnementsystemet, så tilpasning til dette er vanskelig å generalisere. Fleksibilitet for mediaformatet er helt klart det viktigste, og det som endres raskest innenfor IPTV hvor det er viktigst å ha fleksibilitet. Selv om grensesnittet mot kundedatabase og system for bruksovervåkning/betaling lages i overensstemmelse med et rettighetsspråk, vil man fortsatt ha en integreringsjobb.

Med en slik transparent unicast-løsning (VoD), vil vi ikke kunne kryptere media på forhånd, men må også bruke ressurser på dette på samme måte som med multicast (se avsnittet om ressursbruk 8.4). Beskyttelsesfaktoren vil derimot øke av dette, da man får en ny nøkkel for hver VoD-datastrøm.

8.6 Multicast

Multicast benytter seg av UDP. Det vi må tenke på er at UDP-meldinger ikke har noen garanti for at de kommer frem til mottaker, heller ikke i hvilken rekkefølge de mottas. Løsningen som vi beskriver tar hensyn til dette ved å bruke en krypteringsmetode hvor hver melding er uavhengig av andre meldinger eller rekkefølgen de mottas.

8.6.1 Nøkkelbytte

Krypteringsnøklene i vårt system er som sesjonsnøkler å regne. Det vil si at nøklene kun er gyldig en viss periode. I CTR-modus, som vi benytter oss av, er det viktig å være sikker på at ikke samme initialiseringsvektor benyttes flere ganger med sammen nøkkel. Deler av initialiseringsvektoren vil i CTR-modus være en teller, og nøkkelbytte må skje før denne rekker å gjenta seg selv. Dette er beskrevet i avsnitt 6.2.2.

8.7 Videre arbeid

Vår løsning er ment å være en prototype for å undersøke problemstillinger rundt et konkret system, og er ikke en perfekt løsning for beskyttelse av IPTV under alle omstendigheter. Flere av utfordringene ved beskyttelse av IPTV er tett knyttet opp mot utfordringene som allerede er gjeldende for IPTV uten beskyttelsesmekanismer. Dette gjelder skaleringsproblematikk (for koding eller omkoding av videodata), identifisering av brukere og identifisering av mediadata.

Utgangspunktet er at IPTV-beskyttelsen skal være *usynlig*. Løsningen beskrevet i denne rapporten oppfyller i stor grad dette kravet. Men som vi ser blant annet i kapittel 7, er det høye ressurskrav til datakryptering, spesielt på kryptering av unicast-data (for eksempel VoD). Det man vil kunne tjene mest på er optimalisering av krypteringsprosessen, eller begrense hvor mye data som blir kryptert.

Systemet kan gjøres mer fleksibelt ved å lage en plug-in-løsning for forskjellige krypteringsmetoder. Det lar seg allerede gjøre å bytte ut krypteringsalgoritmer, men for eksperimentering med ulike metoder for selektiv kryptering kan dette være en god løsning.

Kapittel 9

Konklusjon

Vi har laget et system for transparent ende-til-ende beskyttelse av sanntids-videostrømmer, med fokus på uavhengighet av videoavspiller, mediaserver og innholdsformat.

Fordelen med en slik transparent løsning er at beskyttelsesmekanismen kan benyttes uavhengig av distribusjonsformat og annen programvare. Vi har vist at slik transparent beskyttelse for multicast er mulig, men samtidig øker dette kompleksiteten til nettverkskommunikasjonen ved innføring av flere ledd for behandling av datastrømmen.

Beskyttelsesmekanismene krever ressurser både på server- og klientsiden. Vi har sett at kryptering av et begrenset antall multicast-strømmer er mulig uten spesielle maskinressurser. På en annen side skalerer sanntidskryptering av unicast/video-on-demand svært dårlig, da ressursbruken på serveren øker lineært med antall klienter. Vårt forslag er å begrense den kryptografiske sikkerheten til et minimum ved å ta i bruk metoder for selektiv kryptering. Fokus på en transparent løsning gjør det derimot vanskeligere å benytte seg av slike metoder.

De videre utfordringene ligger i skalering for en større brukermengde og integrasjonen mot eksisterende administrative løsninger. Det viktigste av alt er at beskyttelsen ikke går i veien for brukeropplevelsen. Løsningen som er lagt frem i rapporten legger til rette for fleksibilitet og fornybarhet i forhold til komponentene i IPTV-systemet, som gjør det mulig å holde følge med den teknologiske utviklingen innenfor IPTV.

Del V

Vedlegg

Tillegg A

Ordforklaring

Analog
Beskriver en enhet eller et system som ikke måles i kun 0 og 1 (digitalt), men som er fysisk varierende og variabelt, f.eks. verden rundt oss.

Big-endian
Beskriver hvordan rekkefølgen på bytes er lagret i minne. En big-endian maskin (f.eks. med RISC prosessor) lagrer den mest signifikante byte på den laveste minneadressen. Se også *Little-endian*.

CA (1)
Conditional Access, tilgangsstyring i MPEG.

CA (2)
Certificate Authority i forbindelse med PKI.

Codec
Forkortelse og sammensetning av *coder* og *decoder*. I audio- og videosammenheng kalles komponenten som komprimerer dataene for en *coder*, og komponenten som regenererer det originale signalet kalles en *decoder*.

CRL
Certificate Revocation List. Holder rede på sertifikater i et PKI-system som er kjent ugyldige og trukket tilbake.

DCT koeffisient
DCT står for Descrete Cosinus Transformation. I MPEG blir dette brukt til å omforme hver bildedel til en verdi.

DRM
Digital Rights Management, Digital Restrictions Managment, Dine Rettig-

heter Minker.

DSL
Digital Subscription Line. Bredbåndslinje.

EPG
Electronic Program Guide. En betegnelse på en tjeneste som gjør TV-programmet tilgjengelig via skjermen. Brukes i satellitt-, kabel-TV-, og IPTV-mottakere. EPG-data blir sendt ut sammen med TV-signalene og brukeren kan navigere seg gjennom TV-programmet og lese detaljer om tilgjengelige programmer og filmer.

H.263
Video codec for videokonferanser med lav bitrate.

Hemmelig nøkkel
Vi bruker hemmelig-/symetrisknøkkel om symetriske nøkler som er kjent for begge parter i en kryptert kommunikasjon. På engelsk brukes ofte *secret key*. Må ikke forveksles med privat nøkkel.

IETF
Internet Engineering Task Force. Utvikler og fremmer Internett standarder.

IP
Internet Protocol. Protokollen (overføringsteknologien) som er brukt til det vi i dag kjenner som Internett.

Little-endian
Beskriver hvordan rekkefølgen på bytes er lagret i minne. En little-endian maskin (f.eks. med x86 prosessor) lagrer den minst signifikante byte på den laveste minneadressen. Se også *Big-endian*.

MD5
Message-Digest algorithm 5. Kryptografisk hash algoritme. Svakheter med algorimen ble publisert i august 2004.

MTU
Maximum Transmission Unit. Maksimal størrelse på et nettverkdatagram. Pakker større enn dette må fragmenteres hvis dette tillates av nettverks-protokollen. IP er en protokoll som tillater fragmentering med mindre det eksplisitt er oppgitt i IP-headeren at dette ikke er tillatt.

nonce
(number used once) En unik engangsverdi brukt i kryptografi. Ofte et til-

feldig generert tall. Kan f.eks. brukes som startverdi for krypterings- eller hash-algoritmer.

Offentlig nøkkel
Dette er det vi på engelsk kaller *public key* innenfor public-key kryptografi. Denne delen av nøkkelen trenger ikke holdes hemmelig.

PKI
Public Key Infrastructure

Privat nøkkel
Dette er det vi på engelsk kaller *private key* innenfor public-key kryptografi. Denne delen av nøkkelen må holdes hemmelig.

RSA
Krypteringsalgoritme, basert på at faktorisering av store tall er vanskelig.

Scrambling
Annet engelsk navn på kryptering/forvrengning av data, ofte brukt i forbindelse med kryptering av TV-signaler. Uttrykket benyttes som regel om enkle (og ofte usikre) analoge eller digitale krypteringsmetoder.

Stream
En strøm av for eksempel lyd eller bilde, der du kontinuerlig blir matet med nok innhold til å spille av uten hakking.

XrML
eXtensible rights Markup Language.

Tillegg B

Brukerveiledning

Dette kapittelet er en kort brukerveiledning som beskriver hvordan man kompilerer og kjører den utviklede prototypen.

B.1 Systemkrav

Operativsystem

Prototypen er utviklet og testet på et GNU/Linux-system, hovedsaklig Debian Linux (testing) og Fedora Core 3. Et GNU/Linux-system med oppdaterte programvarebiblioteker bør være kompatibelt.

På Windows-plattformen vil det i teorien være mulig å kompilere systemet ved hjelp av Cygwin[1] eller MinGW[2].

Programbiblioteker

- GCC. Kompileres med GNU kompilatoren. Versjon 3.3 er benyttet, men er ikke påkrevd.
- Benytter seg av C++ STL. Alternativ kompilator til GCC må støtte dette.
- GNU Common C++. C++ klasser for blant annet flertrådsprogrammering Versjon $\geq$ 1.2.
- libxml2. XML-parsing. Versjon $\geq$ 2.6.
- gcrypt. Symmetriske krypteringsalgoritmer. Versjon $\geq$ 1.3.
- gnutls. TLS-kommunikasjon. Versjon $\geq$ 1.0.

[1] Cygwin: http://www.cygwin.com
[2] MinGW: http://www.mingw.org

Maskinvare

Standard PC-maskinvare med x86-arkitektur. Systemet er utviklet på maskiner med Pentium III 1 GHz prosessor. Se også avsnitt 8.4 som omhandler ressursbruken til klient- og servermodulen.

Nettverkskort må være konfigurert for multicast-kommunikasjon (som regel er dette standard).

B.2 Kompilering

Det forutsettes et POSIX-kompatibelt system. Hvis vi tar utgangspunkt i filen *vldrm-0.1.tar.gz* (se vedlegg G), som er et uttrekk fra CVS, må følgende kommandoer kjøres fra en kommandolinje i for eksempel et *bash-skall*:

```
$ tar zxvf vldrm-cvs.tar.gz
$ cd vldrm
$ ./bootstrap
$ ./configure
$ make
```

Disse kommandoene pakker ut arkivet, sjekker systemkonfigurasjonen og kompilerer kildekoden. *Autoconf*-skriptet *configure* sjekker at alle avhengigheter er tilfredsstilt (programvarebiblioteker, header-filer og versjonsnummer).

For å kjøre enhetstestene kan følgende kommando kjøres fra katalogen *vldrm*:

```
$ make check
```

B.3 Installasjon

Etter at systemet er kompilert kan følgende kommando kjøres for å installere prototypen på systemet:

```
$ make install
```

Dette anbefales foreløpig ikke, fordi applikasjonene ikke er testet utenfor kildekodekatalogen og plasseringen av noen avhengige konfigurasjonsfiler ikke er tilpasset en slik installasjon på dette stadiet.

Sertifkatgenerering

En server trenger en egen CA som signerer alle klientsertifikatene. Slik lager man den selvsignerte CAen ved hjelp av GnuTLS sitt program *certtool*. Det er bare å følge instruksjonene som kommer opp når du starter programmet.

```
$ certtool --generate-privkey --outfile cakey.pem
$ certtool --generate-self-signed --load-privkey cakey.pem \
--outfile ca.pem
```

CAen ligger da i filen `ca.pem` med sin private nøkkel i `cakey.pem`.

For hver klient lages en privatnøkkel, en sertifikatforespørsel og til slutt en signering av sertifikatet:

```
$ certtool --generate-privkey --outfile key.pem
$ certtool --generate-request --load-privkey key.pem \
--outfile request.pem
$ certtool --generate-certificate --load-request request.pem \
--outfile cert.pem --load-ca-certificate ca.pem \
--load-ca-privkey cakey.pem
```

Når dette er ferdig kan `request.pem` (forespørselen) slettes, klientens sertifikat ligger i `cert.pem` og privat nøkkel i `key.pem` Filene som må ligge ved hver klient er: `ca.pem`, `cert.pem` og `key.pem`. Filene som må ligge på serversiden er: `ca.pem` og `cakey.pem`.

B.4 Bruk

Dette avsnittet beskriver en fremgangsmåte for å starte alle komponenter som trengs for å teste systemet fra ende til ende. Det forutsettes i denne beskrivelsen at alle komponentene kjøres på samme maskin. Hvis for eksempel klient- og server-modulen skal kjøre på hver sin maskin, må dette eksempelet endre adresser for å reflektere dette.

Mediastrøm

For å distribuere en kryptert multicast-strøm må man først sette opp en kildestrøm som server-modulen kan prosessere. Vi bruker her applikasjonen *vlc* for å sende en unicast-strøm til lokal UDP port 1235.

```
$ vlc -L --sout udp:localhost:1235 /katalog/til/mediafil.mpg
```

vldrmd

Servermodulen (*vldrmd*) startes fra kommandolinjen slik:

```
$ ./vldrmd
```

Server-modulen er klar til å motta forespørsler fra en eller flere klientmoduler så snart TLS-serveren har generert noen parametere. Dette tar et par sekunder.

For oppsett av multicast-strømmer se et eksempel på en konfigurasjons fil i avsnitt B.6. Definisjon på gyldig syntaks for denne filen er beskrevet ved hjelp av XML Schema (se vedlegg G for filen `vldrmd.xsd`).

Tabell B.1 beskriver de forskjellige oppstartsvalg applikasjonen kan tolke.

Kortvalg	Langvalg	Beskrivelse
-c	–config=FILE	Bruk konfigurasjonsfilen FILE
	–conftest	Validér konfigurasjonsfilen mot DTD-fil
	–conf-schema=FILE	Validér konfigurasjonsfilen mot XML Schema
-s	–syslog	Skriv loggmeldinger til systemloggen
-l	–logfile=FILE	Skriv loggmeldinger til filen FILE
-d	–daemon	Kjør programmet i bakgrunnen
-n	–debug=0-7	Bestem debug-nivå (0-7, 3 er standard, dvs. error)
-v	–version	Skriv ut versjonsnummer og avslutt
-h	–help	Skriv ut hjelpetekst for programmet

Tabell B.1: Kommandolinjevalg for vldrmd

vldrm

Klientmodulen (*vldrm*) startes fra kommandolinjen slik:

```
$ ./vldrm
```

Denne vil nå være klar til å motta forespørsler fra en avspiller.

Tabell B.2 beskriver de forskjellige oppstartsvalg applikasjonen kan tolke.

Kortvalg	Langvalg	Beskrivelse
-s	–server	IP-adressen til servermodulen
-t	–tlsport	TLS-port nummer til servermodulen
-p	–port	Port for lokal HTTP-proxy
-d	–daemon	Kjør programmet i bakgrunnen
-n	–debug=0-7	Bestem debug-nivå (0-7, 3 er standard, dvs. error)
-v	–version	Skriv ut versjonsnummer og avslutt
-h	–help	Skriv ut hjelpetekst for programmet

Tabell B.2: Kommandolinjevalg for vldrm

Avspiller

Avspilleren aksesserer, som tidligere beskrevet, klientmodulen igjennom HTTP-protokollen. Her vises et eksempel hvor avspilleren *vlc* startes fra kommandolinjen mot klientmodulen på port 8888. Multicast-strømmen som skal avspilles har adressen 232.0.0.1:

```
$ vlc http://localhost:8888/MULTICAST/232.0.0.1
```

Alle avspillere som kan spille av en strøm direkte fra enn HTTP-server kan brukes. Unntaket er avspillere som bruker flere tråder for å aksessere serveren.

B.5 Andre merknader

Applikasjonen er konfigurert til å kompilere med debug-informasjon som standard. Kommandoen `--disable-debug` kan brukes for å kompilere uten denne informasjonen. Det anbefales at det kompileres med debug-informasjon fordi det ikke er gjort noen omfattende testing med dette valget skrudd av, og fordi dette vil gjøre det lettere å følge med på hva som skjer under kjøring og testing.

I tillegg til å kompilere med debug-informasjon, kan man kompilere med profileringsinformasjon ved å legge til valget `--enable-profiling` når man kjører *configure*-skriptet. Ved kjøring av applikasjonen vil dette generere informasjon som kan analyseres av verktøyet *gprof*. Merk at applikasjonen vil kjører tregere når denne informasjonen genereres. Bruk derfor bare dette valget når profilering er ønsket. Se også avsnitt 7.2.5.

B.6 Server-konfigurasjon

```
0  <?xml version="1.0" encoding="ISO-8859-1"?>
1  <!DOCTYPE vldrmd SYSTEM "vldrmd.dtd">
2  <!--<vldrmd version="1.0" xmlns:xsi="http://www.w3.org/2001/
       XMLSchema-instance"
3    xsi:noNamespaceSchemaLocation="vldrmd.xsd">-->
4  <vldrmd version="1.0">
5    <info>Test configuration.</info>
6    <daemon>0</daemon>
7    <debug>7</debug>
8    <tlsport>5556</tlsport>
9    <status>
10     <enabled>0</enabled>
11     <ip>0.0.0.0</ip>
12     <port>8880</port>
13   </status>
14   <unicast>
15     <ip>vl.phuzz.org</ip>
16     <port>80</port>
17   </unicast>
```

```
18    <multicast>
19      <stream id="ssm_multicast">
20        <enabled>1</enabled>
21        <info>SSM multicast</info>
22        <cipher>aes</cipher>
23        <ciphermode>ctr</ciphermode>
24        <source>
25          <info>UDP stream 1</info>
26          <ip>0.0.0.0</ip>
27          <port>1235</port>
28        </source>
29        <destination>
30          <info>SSM address</info>
31          <ip>232.0.0.1</ip>
32          <port>1234</port>
33          <ttl>1</ttl>
34        </destination>
35      </stream>
36      <stream id="asm_multicast">
37        <enabled>1</enabled>
38        <info>ASM multicast</info>
39        <cipher>none</cipher>
40        <ciphermode>none</ciphermode>
41        <source>
42          <info>UDP stream 2</info>
43          <ip>0.0.0.0</ip>
44          <port>1236</port>
45        </source>
46        <destination>
47          <info>ASM address</info>
48          <ip>239.0.0.1</ip>
49          <port>1234</port>
50          <ttl>1</ttl>
51        </destination>
52      </stream>
53    </multicast>
54  </vldrmd>
```

Tillegg C

Eksperimenter

Vi har utført noen mindre eksperimenter utenom vår prototype. Dette har vært både for å teste ut måter å løse problemer på, prøve teorier i praksis, og for å kunne måle konkrete resultater for å blant annet kunne regne på ressursbruk.

C.1 SSH overhead og prosessorlast

Vi utførte en del eksperimenter med enkel SSH-tunnelering av videostrømmer for å se på ressursbruk ved kryptering på et tidlig tidspunkt i prosjektet vårt. SSH lager en ende-til-ende forbindelse, og datakrypteringen vil være sammenlignbart med en kryptering av VoD i unicast med bruk av naiv kryptering.

Resultatene fra testkjøringer med dette var noe varierende, men ikke overraskende. Vi brukte *tcpdump* til å overvåke overføringen av en liten videofil. Resultatet fra en typisk gjennomkjøring besto da av mange linjer som i figur C.1 fra *tcpdump*. Disse prosesserte vi med et enkelt Perl-skript (se avsnitt C.1.1) for å trekke ut informasjon gitt i tabell C.1.

```
15:46:40.511415 IP stud253.idi.ntnu.no.www > \
stud255.idi.ntnu.no.42002: P 1449:2433(984) ack 424 \
win 6432 <nop,nop,timestamp 1218140948 1047598548>
```

Figur C.1: Format på *tcpdump*-output.

Vårt opprinnelige mål med eksperimentet var å se om det var noen økt bruk av båndbredde i overføringen av de krypterte dataene. Dette viste seg å bare være marginalt mer. Noe som kan skyldes unøyaktigheter ved bruk av *tcpdump* til å måle overført datamengde.

Det som derimot var interessant, var å se prosessorlasten som var nødvendig for å kryptere videofilen gjennom SSH. Skaleringen av prosessoren er

et viktig punkt som blir kritisk når man innfører sanntidskryptering i tillegg til prosessorlasten som koding av MPEG-data allerede tar.

Fil	Størrelse	Differanse	Pakkestørrelse
original	179 892 224 bytes		
ukryptert	179 892 575 bytes	315 bytes	16384, MTU 16435
kryptert	180 511 776 bytes	619 552 bytes	1448, MTU 1500

Tabell C.1: Resultat fra SSH-forsøket.

C.1.1 *tcpdump* resultat

Programkoden leser fra standard input, plukker ut tallet for brukerdata som ble overført, summerer disse tallene og skriver ut summen til standard output. Programmet forventer at innholdet er formatert som resultatet av en *tcpdump*-kjøring.

```
0 #!/usr/bin/perl -w
1
2 $tot = 0;
3 $count = 0;
4 while (<>) {
5     /^.*IP.*\((\d+?)\).*$/;
6     $tot += $1 if ( $1 );
7     $count++ if ( $1 );
8 }
9 print "$tot ($count)\n";
```

C.2 Lokal nettverks-benchmark

I forbindelse med valg av kommunikasjonsmetode mellom mediaavspiller og klientmodul, foretok vi en benchmark for et par mulige IPC[1]-metoder. Målet med denne testen var å se om TCP kunne fungere som en lokal datakanal for overføring av videodata. For å sammenligne lokal ytelse på TCP-kanalen testet vi også hastigheten ved overføring av data gjennom en *UNIX domain socket*. For å måle overføringshastigheten brukte vi fire enkle programmer som tar tiden på overføring av 256 megabytes med data med en bufferstørrelse på 8192 bytes:

- `tcp_client.c` og `tcp_server.c`. Enkel TCP-klient og server.
- `local_client.c` og `local_server.c`. Enkel UNIX domain socket klient/server.

Testen ble foretatt på en Pentium III 1 GHz maskin. Resultatet vises i tabell C.2 hvor vi kan se av *UNIX domain sockets* har omtrent dobbel så høy ytelse,

[1] Inter Process Communication

men TCP har fortsatt en høy ytelse som er langt over hva som trengs for å overføre videodata.

	Tid (s)	Hastighet (MB/s)
TCP	2,6315	97,2816
UNIX	1,2810	199,8724

Tabell C.2: Resultat fra lokal nettverks-benchmark.

C.3 Chiffer benchmark

For å sette teori ut i praksis satt vi opp en test av de forskjellige blokkchifferalgoritmene som er implementert i *gcrypt*. Dette ble gjort ved å sette opp et minimalt program som benytter seg av gcrypt-biblioteket. Det hverken leses fra eller skrives til disk i løpet av testen, så tallene representerer hvor fort hver algoritme behandler dataene i minne. Dataene som krypteres er randomisert i en bufferstørrelse på 8192. Tidsmålingen er gjort rundt selve krypteringsløkken, så eventuelle oppstartstider og forhåndsberegninger gjort av biblioteket blir ikke tatt med. Chifferalgoritmene vi testet var AES, CAST5, DES, Serpent, Twofish og XTEA i CBC, CFB, CTR og ECB modus (se avsnitt 2.7).

Se avsnitt C.3.1 for kildekoden til løkken som testes, kontrollskriptet, samt datagrunnlaget for testen.

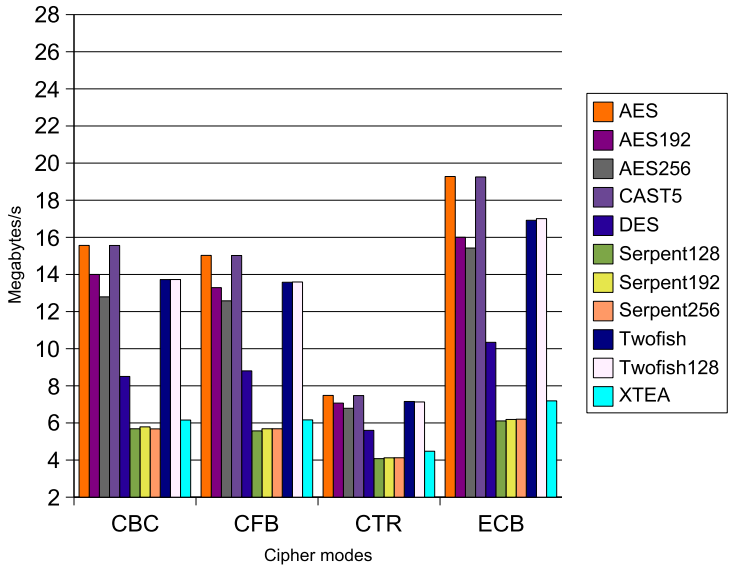
Figurene C.2, C.3 og C.4 viser resultatet fra tre forskjellige maskiner. Det er interessant å se hvordan ytelsen på de forskjellige algoritmene endrer seg mellom prosessortypene. Testene for Pentium 4 og AMD kan til en viss grad sammenlignes. For AMD ser vi en betydelig bedre ytelse av på fleste algoritmene, unntatt AES. CAST5 og Twofish kommer bedre ut enn AES på AMD. Vi kan også se at den tidligere standarden DES ligger langt bak de nye blokkchifferene.

Ut i fra denne testen kan vi konkludere med at algoritmene AES, Twofish og CAST5 er alle aktuelle for vårt formål, hvor høy ytelse er svært viktig, og kanskje også viktigere enn å sikre dataene på best mulig måte. Siden AES er den mest standardiserte algoritmen, og begynner å få stor utbredelse, vil vi bruke denne som standard for prototypen vår.

Hvilken modus man velger er ikke like opplagt bare ut i fra denne testen. ECB har sine klare svakheter som er diskutert tidligere i rapporten, og CBC er best egnet til kryptering av filer på et statisk lager. Alternativene er derfor CFB og CTR. Ved bruk av CFB må IV-en for hver melding legges ved i en header og øke størrelsen på hver pakke. Det antas at CFB har en segmentstørrelse på 8 bits (1 byte) i gcrypt-biblioteket. Dette er ikke beskrevet noe sted og er noe vanskelig å se ut i fra implementasjonen. Men ved eksperimentering fungerer den på meldinger av vilkårlig byte-størrelse.

Den modusen man da står igjen med er CTR. I følge [Sch96] er CTR-modus sterk anbefalt, men som vi ser på testen kommer denne dårlig ut. Vi antar at dette skyldes implementasjonen i gcrypt-biblioteket. Ved nærmere undersøkelse av implementeringen av de forskjellige chiffermodi ser det ut som CTR-implementasjonens bruk av en modulusoperasjon (divisjonsrest: %) står for det meste av tidstapet. Et uavhengig testprogram tok tiden på en løkke med en enkel modulusoperasjon og sammenlignet tiden for den samme løkken med en addisjonsoperasjon[2]. Hastigheten til modulusløkken var omtrent 70% av addisjonsløkken. Dette er også omtrent forholdet mellom CBC/CFB og CTR-modusene.

Fordelen med CTR er at header-størrelsen kan reduseres ved å utelate deler av IV-eni, så lenge man har nok informasjon som kan brukes til å regne seg frem til IV-en hos mottaker.

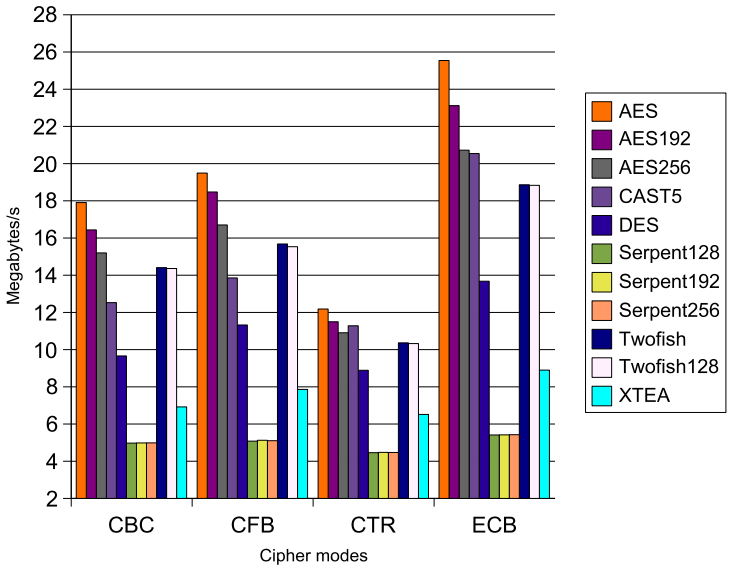


Figur C.2: gcrypt cipher benchmark, Pentium 3 1GHz, 64MB

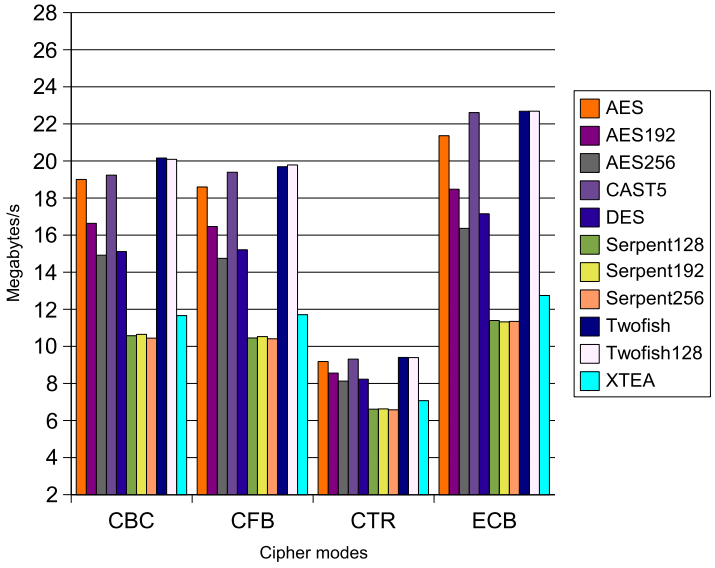
C.3.1 Datagrunnlag

Tabellene C.3, C.4 og C.5 viser datagrunnlaget for figurene C.2, C.3 og C.4.

[2]Addisjon ble valgt for å ha en operasjon å sammenligne med, og fordi den antas å være raskerer enn modulusoperasjonen.



Figur C.3: gcrypt cipher benchmark, Pentium 4 1.8GHz, 64MB



Figur C.4: gcrypt cipher benchmark, AMD XP 1800+, 64MB

Cipher	Mode	time	B/s	AES%	Mb/s
AES	ECB	3,319780	20 214 852	100,00	19,28
AES	CTR	8,552434	7 846 756	100,00	7,48
AES	CBC	4,111466	16 322 368	100,00	15,57
AES	CFB	4,258476	15 758 892	100,00	15,03
AES192	ECB	3,998586	16 783 148	120,45	16,01
AES192	CTR	9,053160	7 412 755	105,85	7,07
AES192	CBC	4,570616	14 682 673	111,17	14,00
AES192	CFB	4,815374	13 936 376	113,08	13,29
AES256	ECB	4,148642	16 176 103	124,97	15,43
AES256	CTR	9,423686	7 121 296	110,19	6,79
AES256	CBC	5,002690	13 414 555	121,68	12,79
AES256	CFB	5,087650	13 190 542	119,47	12,58
Twofish	ECB	3,782314	17 742 806	113,93	16,92
Twofish	CTR	8,940288	7 506 342	104,54	7,16
Twofish	CBC	4,662934	14 391 982	113,41	13,73
Twofish	CFB	4,712172	14 241 598	110,65	13,58
Twofish128	ECB	3,762518	17 836 157	113,34	17,01
Twofish128	CTR	8,971816	7 479 964	104,90	7,13
Twofish128	CBC	4,661938	14 395 057	113,39	13,73
Twofish128	CFB	4,707656	14 255 260	110,55	13,59
Serpent128	ECB	10,476280	6 405 791	315,57	6,11
Serpent128	CTR	15,699200	4 274 667	183,56	4,08
Serpent128	CBC	11,252500	5 963 907	273,69	5,69
Serpent128	CFB	11,483160	5 844 111	269,65	5,57
Serpent192	ECB	10,339900	6 490 281	311,46	6,19
Serpent192	CTR	15,522760	4 323 255	181,50	4,12
Serpent192	CBC	11,052560	6 071 793	268,82	5,79
Serpent192	CFB	11,250620	5 964 903	264,19	5,69
Serpent256	ECB	10,315460	6 505 658	310,73	6,20
Serpent256	CTR	15,510660	4 326 628	181,36	4,13
Serpent256	CBC	11,265200	5 957 183	273,99	5,68
Serpent256	CFB	11,251980	5 964 182	264,23	5,69
XTEA	ECB	8,901352	7 539 176	268,13	7,19
XTEA	CTR	14,295880	4 694 280	167,16	4,48
XTEA	CBC	10,392060	6 457 705	252,76	6,16
XTEA	CFB	10,381920	6 464 012	243,79	6,16
DES	ECB	6,186058	10 848 405	186,34	10,35
DES	CTR	11,426800	5 872 935	133,61	5,60
DES	CBC	7,523260	8 920 184	182,98	8,51
DES	CFB	7,267700	9 233 851	170,66	8,81
CAST5	ECB	3,323688	20 191 084	100,12	19,26
CAST5	CTR	8,561452	7 838 490	208,23	7,48
CAST5	CBC	4,111864	16 320 788	96,56	15,56
CAST5	CFB	4,259956	15 753 417	100,03	15,02

Tabell C.3: gcrypt cipher benchmark, Pentium 3 1GHz, 64MB

Cipher	Mode	time	B/s	AES%	Mb/s
AES	ECB	2,505714	26 782 331	100,00	25,54
AES	CTR	5,252734	12 775 987	100,00	12,18
AES	CBC	3,574322	18 775 270	100,00	17,91
AES	CFB	3,283218	20 439 965	100,00	19,49
AES192	ECB	2,768396	24 241 063	110,48	23,12
AES192	CTR	5,565976	12 056 980	105,96	11,50
AES192	CBC	3,893780	17 234 888	108,94	16,44
AES192	CFB	3,464244	19 371 864	105,51	18,47
AES256	ECB	3,088444	21 729 020	123,26	20,72
AES256	CTR	5,868038	11 436 337	111,71	10,91
AES256	CBC	4,210774	15 937 417	117,81	15,20
AES256	CFB	3,831824	17 513 555	116,71	16,70
Twofish	ECB	3,393254	19 777 141	135,42	18,86
Twofish	CTR	6,175378	10 867 166	117,57	10,36
Twofish	CBC	4,441754	15 108 640	124,27	14,41
Twofish	CFB	4,081372	16 442 721	124,31	15,68
Twofish128	ECB	3,398484	19 746 705	135,63	18,83
Twofish128	CTR	6,197486	10 828 401	117,99	10,33
Twofish128	CBC	4,456370	15 059 087	124,68	14,36
Twofish128	CFB	4,120544	16 286 408	125,50	15,53
Serpent128	ECB	11,826320	5 674 534	471,97	5,41
Serpent128	CTR	14,352980	4 675 604	273,25	4,46
Serpent128	CBC	12,865280	5 216 276	359,94	4,97
Serpent128	CFB	12,595940	5 327 817	383,65	5,08
Serpent192	ECB	11,811340	5 681 731	471,38	5,42
Serpent192	CTR	14,297980	4 693 590	272,20	4,48
Serpent192	CBC	12,837680	5 227 491	359,16	4,99
Serpent192	CFB	12,483360	5 375 865	380,22	5,13
Serpent256	ECB	11,790240	5 691 899	470,53	5,43
Serpent256	CTR	14,314140	4 688 291	272,51	4,47
Serpent256	CBC	12,832440	5 229 626	359,02	4,99
Serpent256	CFB	12,540140	5 351 524	381,95	5,10
XTEA	ECB	7,192530	9 330 355	287,05	8,90
XTEA	CTR	9,819748	6 834 071	186,95	6,52
XTEA	CBC	9,246524	7 257 739	258,69	6,92
XTEA	CFB	8,139286	8 245 055	247,91	7,86
DES	ECB	4,679160	14 342 075	186,74	13,68
DES	CTR	7,198146	9 323 076	137,04	8,89
DES	CBC	6,623758	10 131 539	185,32	9,66
DES	CFB	5,651664	11 874 177	172,14	11,32
CAST5	ECB	3,115510	21 540 249	124,34	20,54
CAST5	CTR	5,673466	11 828 547	158,73	11,28
CAST5	CBC	5,108478	13 136 762	155,59	12,53
CAST5	CFB	4,619356	14 527 753	140,70	13,85

Tabell C.4: gcrypt cipher benchmark, Pentium 4 1.8GHz, 64MB

Cipher	Mode	time	B/s	AES%	Mb/s
AES	ECB	2,996222	22 397 827	100,00	21,36
AES	CTR	6,969224	9 629 316	100,00	9,18
AES	CBC	3,367726	19 927 055	100,00	19,00
AES	CFB	3,441242	19 501 349	100,00	18,60
AES192	ECB	3,463906	19 373 754	115,61	18,48
AES192	CTR	7,477778	8 974 439	107,30	8,56
AES192	CBC	3,847598	17 441 755	114,25	16,63
AES192	CFB	3,887268	17 263 760	112,96	16,46
AES256	ECB	3,910596	17 160 776	130,52	16,37
AES256	CTR	7,875510	8 521 208	113,00	8,13
AES256	CBC	4,290226	15 642 267	127,39	14,92
AES256	CFB	4,340172	15 462 259	126,12	14,75
Twofish	ECB	2,821710	23 783 047	94,18	22,68
Twofish	CTR	6,805814	9 860 519	97,66	9,40
Twofish	CBC	3,175026	21 136 476	94,28	20,16
Twofish	CFB	3,250974	20 642 694	94,47	19,69
Twofish128	ECB	2,821382	23 785 812	94,16	22,68
Twofish128	CTR	6,813000	9 850 119	97,76	9,39
Twofish128	CBC	3,185906	21 064 295	94,60	20,09
Twofish128	CFB	3,234938	20 745 023	94,00	19,78
Serpent128	ECB	5,616488	11 948 545	187,45	11,40
Serpent128	CTR	9,678236	6 933 997	138,87	6,61
Serpent128	CBC	6,053186	11 086 535	179,74	10,57
Serpent128	CFB	6,122578	10 960 883	177,92	10,45
Serpent192	ECB	5,652960	11 871 455	188,67	11,32
Serpent192	CTR	9,658914	6 947 868	138,59	6,63
Serpent192	CBC	6,010772	11 164 766	178,48	10,65
Serpent192	CFB	6,078848	11 039 733	176,65	10,53
Serpent256	ECB	5,640044	11 898 641	188,24	11,35
Serpent256	CTR	9,731004	6 896 396	139,63	6,58
Serpent256	CBC	6,126690	10 953 526	181,92	10,45
Serpent256	CFB	6,148502	10 914 668	178,67	10,41
XTEA	ECB	5,021808	13 363 486	167,60	12,74
XTEA	CTR	9,048620	7 416 474	129,84	7,07
XTEA	CBC	5,488578	12 227 003	162,98	11,66
XTEA	CFB	5,465458	12 278 726	158,82	11,71
DES	ECB	3,731862	17 982 675	124,55	17,15
DES	CTR	7,777642	8 628 433	111,60	8,23
DES	CBC	4,234844	15 846 832	125,75	15,11
DES	CFB	4,208350	15 946 597	122,29	15,21
CAST5	ECB	2,830814	23 706 560	94,48	22,61
CAST5	CTR	6,872726	9 764 519	98,62	9,31
CAST5	CBC	3,327526	20 167 795	98,81	19,23
CAST5	CFB	3,300398	20 333 567	95,91	19,39

Tabell C.5: gcrypt cipher benchmark, AMD XP 1800+, 64MB

Kode for gcrypt benchmark testløkke

Denne kodebiten er hentet fra `gcrypt_encrypt.cpp` og viser hvordan tidsmålingen av løkken er foretatt.

```
1      gcry_randomize(plaintext, buflen, GCRY_STRONG_RANDOM);
2
3      double t1, t2;
4      struct timeval tp;
5      gettimeofday(&tp, NULL);
6      t1 = (double)tp.tv_sec + (1.e-6) * tp.tv_usec;
7
8      // benchmark loop
9      for ( nbytes = buflen; totbytes < mbytes && !err;
           totbytes += nbytes ){
10         err = gcry_cipher_encrypt(hd,
11                 ciphertext, nbytes,
12                 plaintext, nbytes);
13     }
14
15     gettimeofday(&tp, NULL);
16     t2 = (double)tp.tv_sec + (1.e-6) * tp.tv_usec;
17     cout << "CPU TIME: " << (t2-t1) << endl;
```

Kontrollskript for gcrypt benchmark

```
0  #!/bin/bash
1  #
2  # Runs a benchmark test on the gcrypt ciphers in different
       modes.
3  #
4
5  MBYTES=64
6  BYTES="$(echo $MBYTES \* 1024 \* 1024 | bc)"
7  CIPHERS="aes aes192 aes256 twofish twofish128 serpent128
       serpent192 serpent256 tea des cast5"
8  MODES="ecb ctr cbc cfb"
9  EXEC="./gcrypt_encrypt"
10
11 if [ ! -x $EXEC ]; then
12     if [ ! -f gcrypt_tea.o ]; then
13         echo "Compiling TEA cipher..."
14         g++ -c gcrypt_tea.cpp -o gcrypt_tea.o
15     fi
16     echo "Compiling executable..."
17     g++ `libgcrypt-config --libs --cflags` gcrypt_encrypt.
           cpp gcrypt_tea.o -o $EXEC
18 fi
19
20 CPU="$(cat /proc/cpuinfo |grep 'model name'|cut -d':' -f2)"
21 CPUSPEED="$(cat /proc/cpuinfo |grep 'cpu MHz'|cut -d':' -f2)
       "
22 CACHE="$(cat /proc/cpuinfo |grep 'cache size'|cut -d':' -f2)
       "
23 echo "# SIZE=${MBYTES}MB CPU=$CPU $CPUSPEED MHz CACHE=$CACHE
       "
24
```

```
25 for cipher in $CIPHERS; do
26     for mode in $MODES; do
27         CPUTIME_TOT=0
28         for count in 1 2 3 4 5; do
29             CPUTIME=$($EXEC $cipher $mode $MBYTES | \
30                 grep 'CPU TIME:' | \
31                 cut -d':' -f2)
32             CPUTIME_TOT=$(echo $CPUTIME_TOT + $CPUTIME | bc
                   -l)
33         done
34         CPUTIME_AVG="$(echo $CPUTIME_TOT \/ $count | bc -l)"
35         BYTES_PER_SEC="$(echo $BYTES \/ $CPUTIME_AVG | bc)"
36         echo "$cipher $mode $CPUTIME_AVG $BYTES_PER_SEC"
37     done
38 done
```

XTEA

XTEA er ikke implementert i gcrypt. Implementasjonen til dette blokkchifferet er hentet fra Internett[3]. Den er deretter registrert som chifferalgoritme i gcrypt-biblioteket slik at dette implementerer de forskjellige chiffermodusene.

Nettsiden som algoritmen er hentet fra beskriver dette som en av de raskeste og mest effektive krypteringsalgoritmer som finnes. Som vi kan se av våre tester (fig. C.2) får ikke vi dette til å stemme, og algoritmen kommer ut som en av de tregeste. Testen er gjennomført med XTEA i 32 runder. Ifølge kommentarene i implementasjonen kan dette rundeantallet senkes til så lite som 8 for video-data. Ved 8 runder[4] havnet algoritmen på en hastighet som kan sammenlignes med DES i ECB modus. Ifølge [WN94] er TEA ment å være et alternativ til DES og beskrives som opptil tre ganger raskere enn DES ved 32 runder (DES har 16 runder). Vi tror hastighetsforskjellene i vår test skyldes at chifferalgoritmene i gcrypt er godt optimalisert med blant annet mange forhåndsberegninger.

Siden denne algoritmen i utgangspunktet virket interessant, dobbeltsjekket vi resultatet ved å lage en enklere test-implementasjon uten bruk av gcrypt. Resultatet viste kun en 5% forbedring i forhold til gcrypt-testen. Se avsnitt C.3.2 for kildekode og kommentarer til denne testen.

Selv om algoritmen ikke når helt opp ytelsesmessing, så kan den fortsatt være interessant for eksempel i forbindelse med set-top-bokser med lite ressurser. Algoritmen er portabel og lett å implementere. Det finnes blant annet tilgjengelig kildekode i assembly for ulike prosessorer.

[3] TEA: http://www.simonshepherd.supanet.com/tea.htm

[4] Av grunner vi ikke har gått i dybden på så ville ikke algoritmen dekryptere dataene korrekt da vi endre rundeantallet til noe annet enn 32.

C.3.2 XTEA

Denne testen kjører XTEA chifferet i ECB modus på 64 megabytes med data. Programmet er brukt til å teste den rå hastigheten til chifferet. Resultatet var bare marginalt bedre enn ved å registrere chifferet i gcrypt (ca. 5% raskere).

tea_benchmark.cpp

```
0  #include <iostream>
1  #include <sys/time.h>
2  #include <ctime>
3  #include "gcrypt_tea.h"
4
5  using namespace std;
6
7  /**
8   * Benchmark the XTEA cipher in ECB mode.
9   */
10 int main(){
11     unsigned char text[gcrypt_tea::BLOCKSIZE + 1];
12     unsigned char ciphertext[gcrypt_tea::BLOCKSIZE + 1];
13     unsigned char plaintext[gcrypt_tea::BLOCKSIZE + 1];
14     unsigned char key[gcrypt_tea::KEYLENB + 1];
15     unsigned int nbytes, totbytes, mbytes = 64*1024*1024;
16     gcrypt_tea::CONTEXT ctx;
17
18     memcpy(key, "0123456789ABCDEF", gcrypt_tea::KEYLENB);
19     gcrypt_tea::setkey(&ctx, key, gcrypt_tea::KEYLENB);
20     memcpy(text, "SECRET\0\0", gcrypt_tea::BLOCKSIZE);
21
22     double t1, t2;
23     struct timeval tp;
24     gettimeofday(&tp, NULL);
25     t1 = (double)tp.tv_sec + (1.e-6) * tp.tv_usec;
26
27     // benchmark loop
28     for ( nbytes = gcrypt_tea::BLOCKSIZE, totbytes = 0;
29             totbytes < mbytes; totbytes += nbytes ){
30         gcrypt_tea::encipher(&ctx,
31                 (unsigned long*)text,
32                 (unsigned long*)ciphertext,
33                 (unsigned long*)key);
34     }
35
36     gettimeofday(&tp, NULL);
37     t2 = (double)tp.tv_sec + (1.e-6) * tp.tv_usec;
38     cout << (t2-t1) << endl;
39
40     gcrypt_tea::decipher(&ctx,
41             (unsigned long*)ciphertext,
42             (unsigned long*)plaintext,
43             (unsigned long*)key);
44
45     if ( memcmp(text, plaintext, gcrypt_tea::BLOCKSIZE) == 0
           ){
46         return 0;
47     }else{
48         cout << "Failed to decrypt data!" << endl;
49         return 1;
50     }
51 }
```

Tillegg D

Klassediagrammer

D.1 Felles klasser

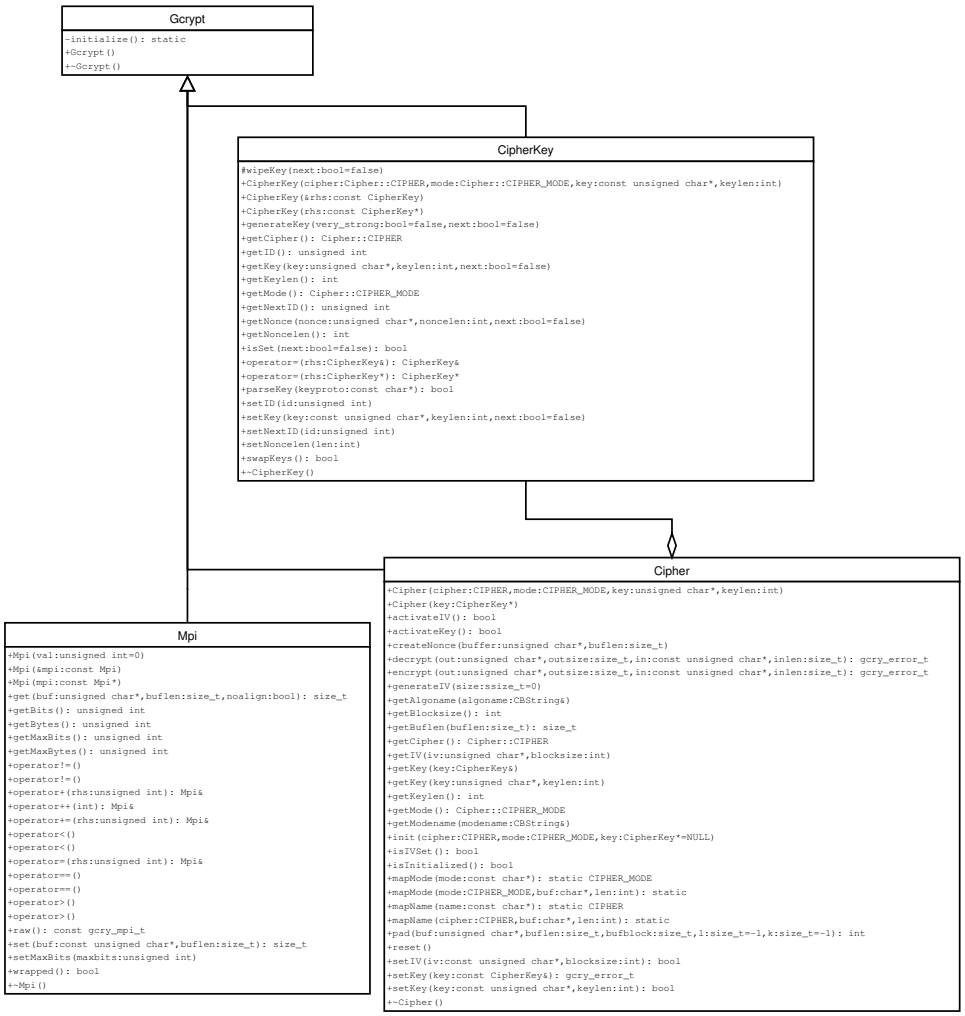
Disse klassene har funksjoner for både i klient- og servermodul.

UDP-leser

Reader er en abstrakt klasse, UdpReader blir brukt til å les fra unicast og multicast på henholdsvis server og klient

Krypteringsfunksjonalitet

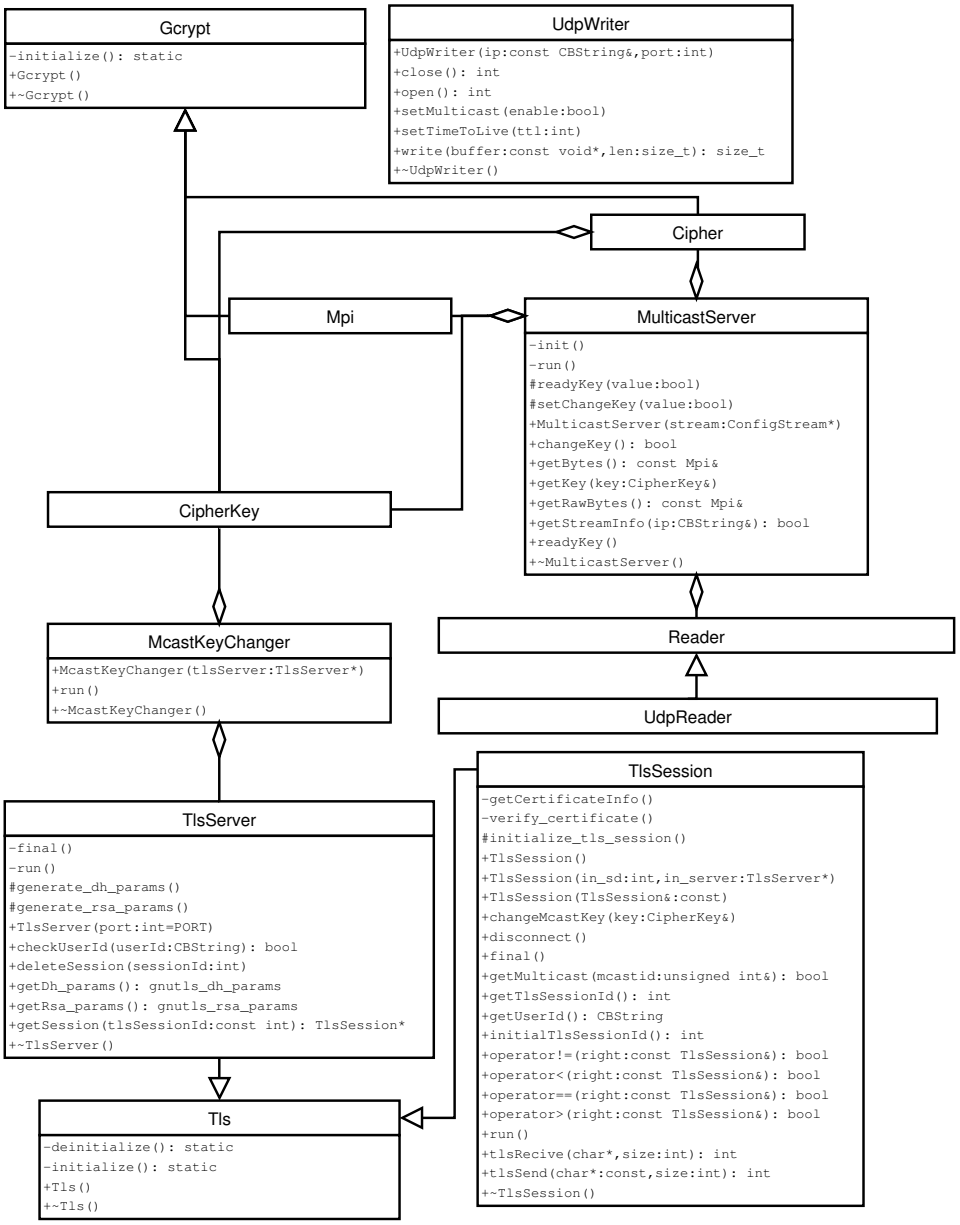
Gcrypt er biblioteket med krypteringsfunksjonaliteten. Mpi er for behandling av lange heltall og blir kun brukt på serveren.



D.2 Serverklasser

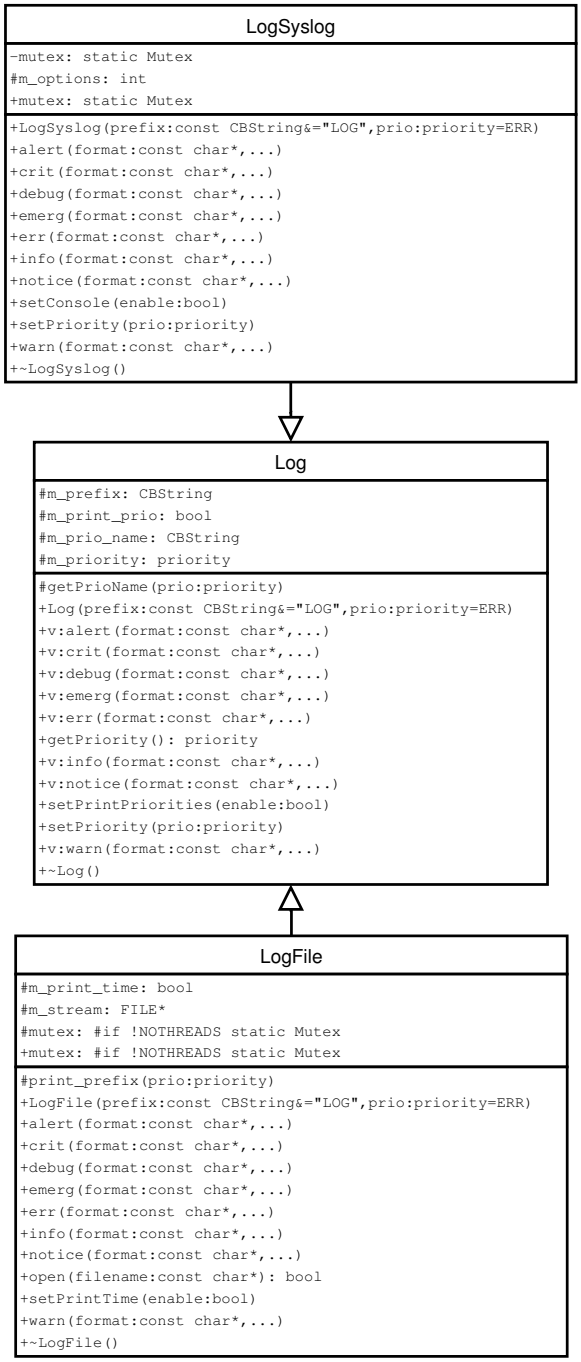
Klasser som er brukt i servermodulen.

Serverdiagram



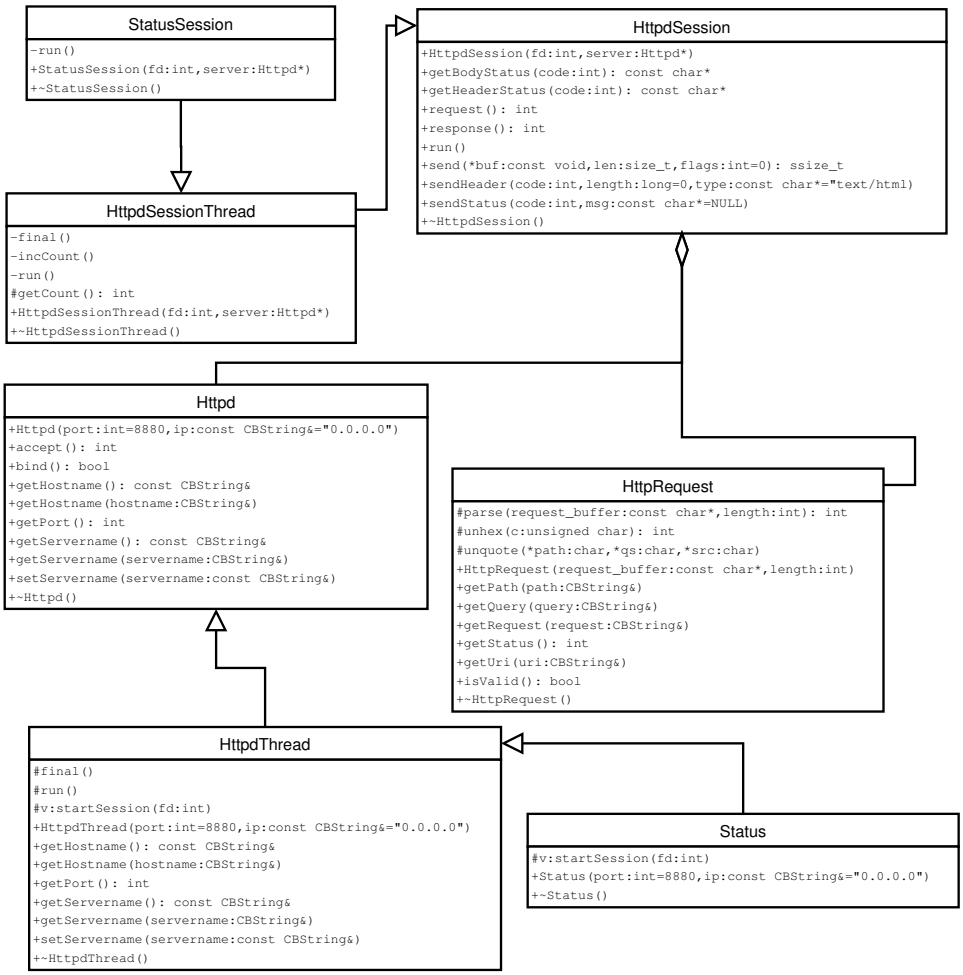
Logging

Disse klassene har loggfunksjonaliteten som kan skrive til fil, skjerm eller syslog etter behov.



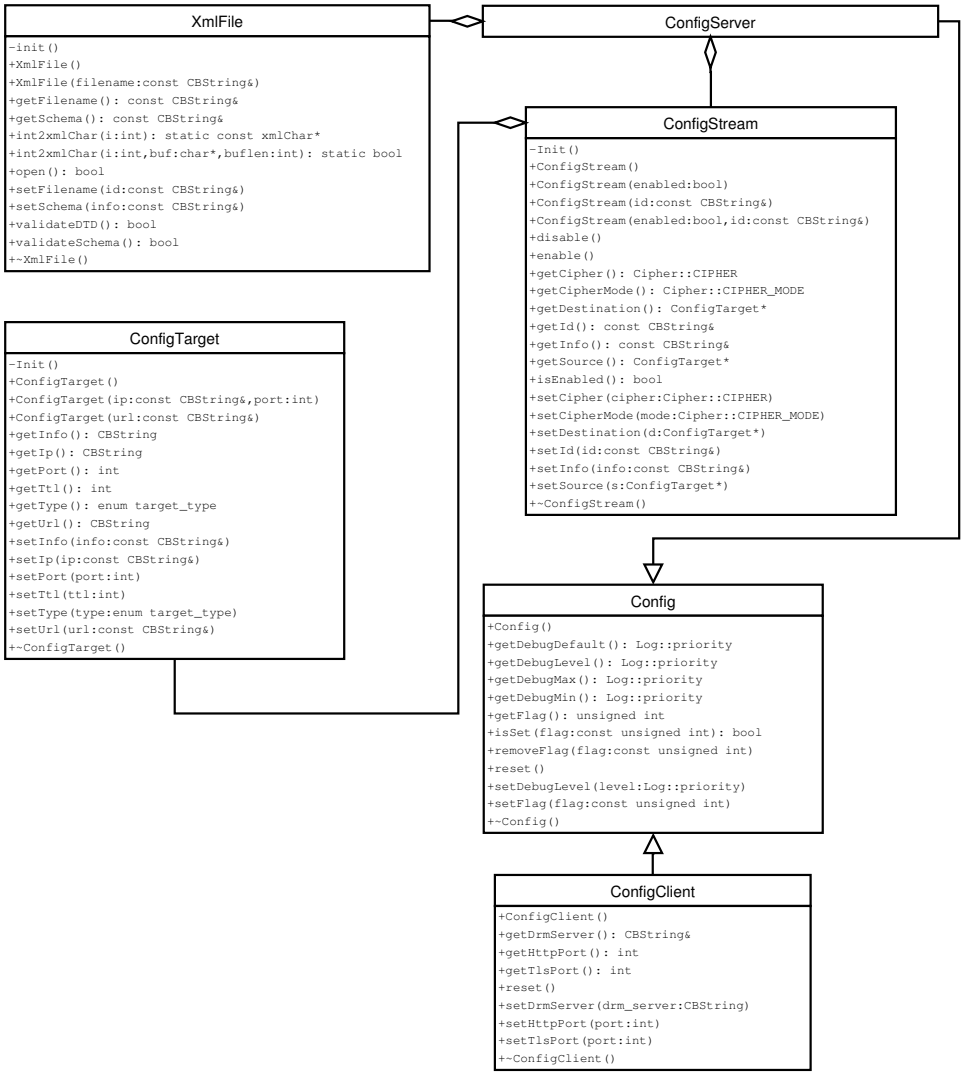
Webserver

Dette er webserverdelen på servermodulen.



Konfigurasjon

Klassene for behandling av konfigurasjon av alle serverparametere.

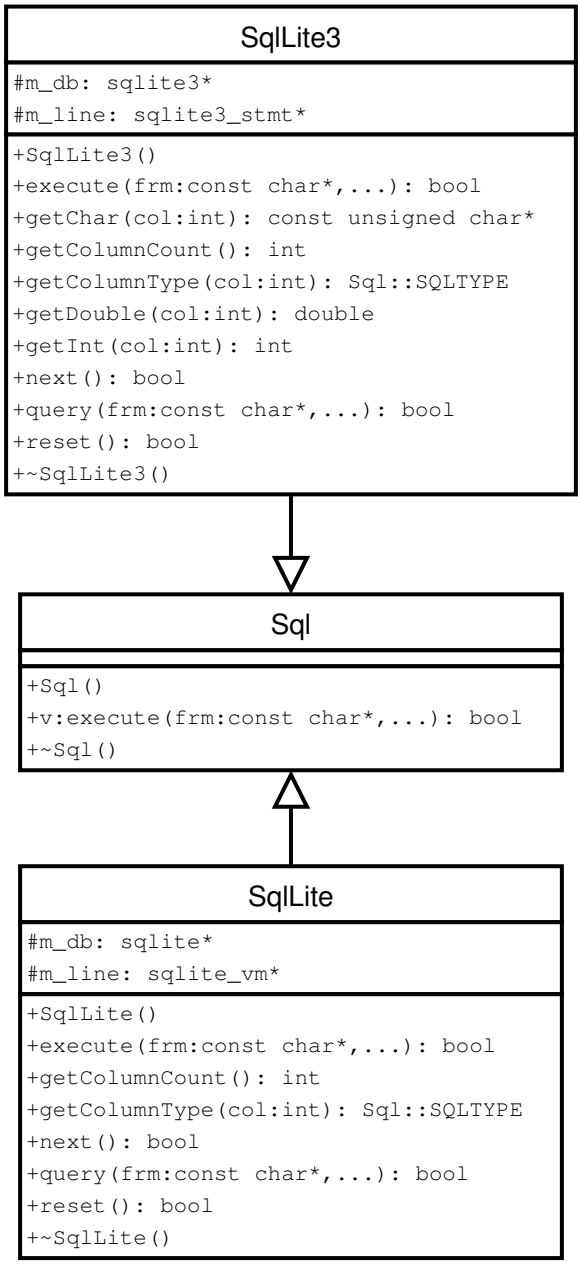


Konfigurasjonsserver

Detaljer for ConfigServer klassen.

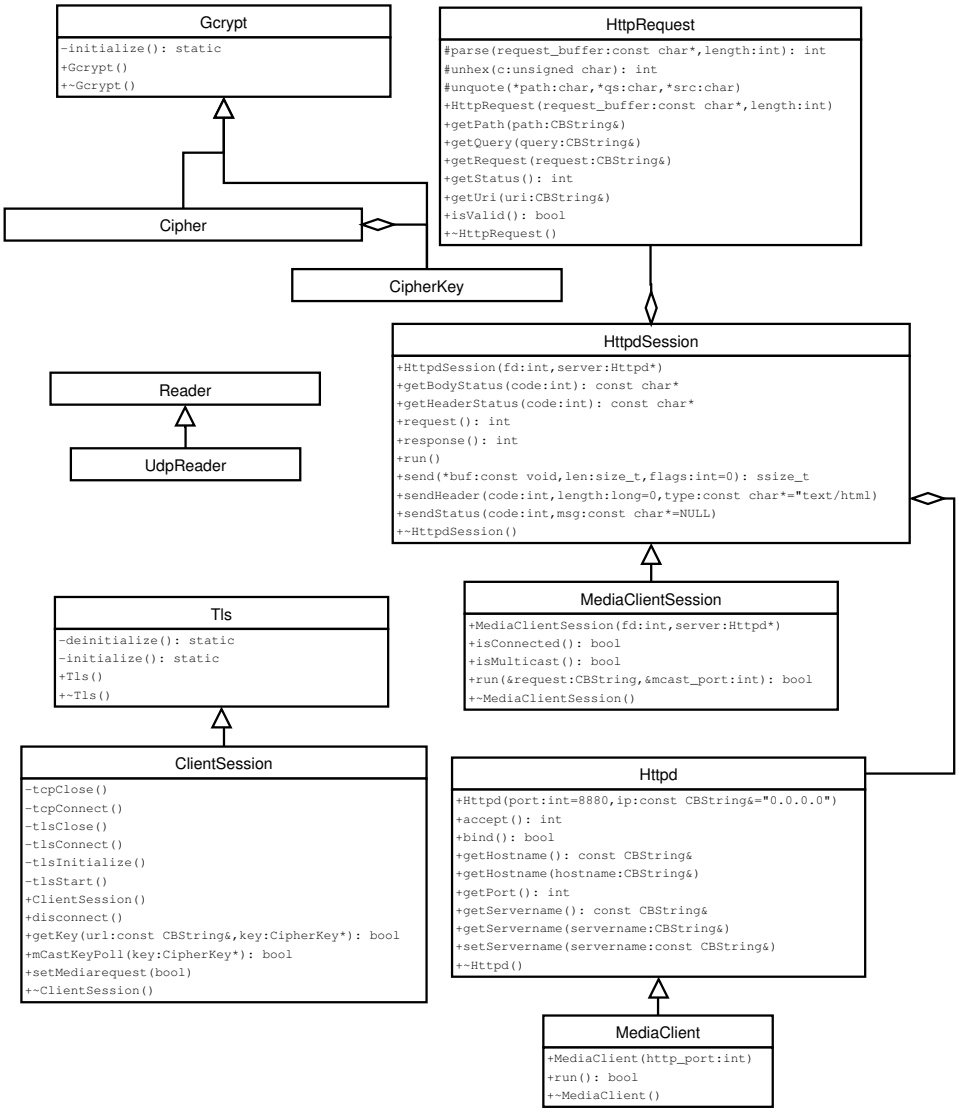
SQL-grensesnitt

Omsluttningsklasser for SQL kommandoer.



D.3 Klientklasser

Klasser for klientmodulen.



Tillegg E

Kildekode

Dette avsnittet inneholder noen viktige utdrag fra kildekoden. Se vedlegg G for all kildekode.

E.1 Multicast-kryptering (server)

Kryptering av multicast på servermodulen (`mcast_server.cpp:125-311`).

```
125 void MulticastServer::run(){
126     INFO("multicast thread '%s' starting", getName());
127 
128     switch ( m_stream->getSource()->getType() ){
129         case ConfigTarget::IP:
130             m_source = new UdpReader(m_stream->getSource()->
                    getIp(),
131                         m_stream->getSource()->getPort());
132             break;
133         case ConfigTarget::URL:
134             ERROR("multicast source type URL not implemented
                    .");
135             return;
136             break;
137         case ConfigTarget::FILE:
138             //m_source = new MpegReader(m_stream->getSource
                    ()->getUrl());
139             WARN("multicast source type FILE not implemented
                    .");
140             return;
141             break;
142         default:
143             ERROR("multicast source type not recognized.");
144             return;
145             break;
146     }
147 
148     Cipher::CIPHER cipher = m_cipher->getCipher();
149     if ( cipher == Cipher::CIPHER_NONE )
150         DEBUGP("multicast stream not protected");
151 
152     for (;;){
```

```
153         // Connect and read from source
154         if ( m_source->open(0) == -1 ){
155             // open failed, retry after 3 seconds
156             NOTICE("multicast source '%s' failed to open",
                    getName());
157             Thread::sleep(3000);
158             continue;
159         }
160 
161         // Read from source and forward as multicast
                packages
162         UdpWriter writer(m_stream->getDestination()->getIp()
                ,
163                 m_stream->getDestination()->getPort());
164         if ( writer.open() == -1 ){
165             // open failed, retry after 3 seconds
166             NOTICE("multicast writer '%s' failed to open",
                    getName());
167             Thread::sleep(3000);
168             continue;
169         }
170         writer.setMulticast(true);
171         writer.setTimeToLive(m_stream->getDestination()->
                getTtl());
172 
173         time_t keytime, nowtime;
174         time(&keytime);
175         unsigned int ivsize = 8;
176         Mpi ivctr;
177         ivctr.setMaxBits(ivsize * 8);
178         Mpi msgCount;
179         msgCount.setMaxBits(64);
180         Mpi genCount;
181         genCount.set((unsigned char*)"\x00\x00\xff\xff\xff\
                xff\xff\xff", 8);
182         unsigned int blocksize = m_cipher->getBlocksize();
183         size_t nbytes;
184         m_iv = new unsigned char[ivsize];
185         m_ivblock = new unsigned char[blocksize];
186         m_buffer = new unsigned char[NETBUFSIZE + sizeof(
                CRYPT_HEADER) + ivsize];
187         unsigned char* buf;
188         CRYPT_HEADER chdr;
189         unsigned int keyid;
190         if ( cipher == Cipher::CIPHER_NONE ){
191             keyid = 0;
192             buf = m_buffer + sizeof(CRYPT_HEADER);
193         }else{
194             keyid = 1;
195             buf = m_buffer + sizeof(CRYPT_HEADER) + ivsize;
196         }
197 
198         // benchmark
199         /*
200         double t1, t2;
201         struct timeval tp;
202         char benchbuf[1024];
203         char benchfile[255];
204         snprintf(benchfile, sizeof(benchfile), "bench_%s.out
```

```
205                 ",
                        (const char*)(m_stream->getId()));
206             FILE* fd = fopen(benchfile, "w");
207             */
208
209             m_source->reset();
210             m_source->sync();
211             chdr.msgId = 0;
212             m_cipher->generateIV(ivsize);
213             m_cipher->getIV(m_iv, ivsize);
214             ivctr.set(m_iv, ivsize);
215
216             while ( (nbytes = m_source->read(buf, NETBUFSIZE)) >
                     0 ){
217                 // benchmark
218                 /*
219                 gettimeofday(&tp, NULL);
220                 t1 = (double)tp.tv_sec + (1.e-6) * tp.tv_usec;
221                 */
222
223                 if ( cipher != Cipher::CIPHER_NONE ){
224                     // generate new key
225                     // key is changes at least every second hour
                            (less if
226                     // debugging)
227                     time(&nowtime);
228 #if DEBUG
229                     if ( !m_changeKey && (nowtime-keytime) > 20
                            || (msgCount > genCount) ){
230 #else
231                     if ( !m_changeKey && (nowtime-keytime) >
                            7200 || (msgCount > genCount) ){
232 #endif
233                         msgCount = 1;
234                         DEBUGP("%s: generate new key(%d) nextkey
                                (%d)",
235                                 getName(), m_key->getID(), m_key
                                    ->getNextID());
236                         m_key->generateKey(false, true);
237                         mutex.enter();
238                         m_changeKey = true;
239                         mutex.leave();
240                     }
241
242                     // change key when key changer has
                            distributed the keys
243                     if ( m_readyKeyChange || msgCount.wrapped()
                            ){
244                         time(&keytime);
245                         // The actual key change is done here
246                         DEBUGP("%s: using new key(%d) nextkey(%d
                                )",
247                                 getName(), m_key->getID(), m_key
                                    ->getNextID());
248                         mutex.enter();
249                         m_key->swapKeys();
250                         m_cipher->activateKey();
251
252                         // new IV
```

```
253                 m_cipher->generateIV(ivsize);
254                 m_cipher->getIV(m_iv, ivsize);
255                 ivctr.set(m_iv, ivsize);
256 
257                 keyid = m_key->getID();
258                 m_readyKeyChange = false;
259                 m_changeKey = false;
260                 mutex.leave();
261             }
262 
263             // CTR IV format:
264             // NNNNIIIIIIIICCCC
265             // N = nonce, I = IV, C = block counter (1)
266             memset(m_ivblock, 0, blocksize);        // IV = 0
267             m_key->getNonce(m_ivblock, m_key->getNoncelen());   // Nonce
268             ivctr.get(m_ivblock + m_key->getNoncelen(), ivsize);// IV
269             m_ivblock[blocksize-1] = 0x01;          // counter = 1
270             m_cipher->setIV(m_ivblock, blocksize);
271 
272             // Pseudo-selective encryption (needs to specify
273             // some parameters in the header for the client
274             // to be able to decrypt).
275             // Encrypts the first 1/4 of a MPEG TS-packet.
276             /*
277             unsigned int segmentlen = 188;
278             unsigned int cryptlen = 47;
279             for ( unsigned int i = 0; i < nbytes; i += segmentlen){
280                 //FIXME, the buffer will overflow here if
281                 //nbytes is not a multiple of segmentlen.
282                 m_cipher->encrypt(buf+i, cryptlen, NULL, 0);
283                 // scramble the rest of the data using the ciphertext
284                 for ( unsigned int j = i+cryptlen; j < i+segmentlen; j++ )
285                     buf[j] ^= buf[j-cryptlen];
286             }
287             */
288             m_cipher->encrypt(buf, nbytes, NULL, 0);
289 
290             ivctr++;    // IV++
291 
292             chdr.keyId = keyid;
293             memcpy(m_buffer, &chdr, sizeof(CRYPT_HEADER));
294             memcpy(m_buffer + sizeof(CRYPT_HEADER),
295                     m_ivblock + m_key->getNoncelen(), ivsize);
296             m_rawbytes += nbytes;
```

```
297                 nbytes += sizeof(CRYPT_HEADER) + ivsize;
298             }else{
299                 chdr.keyId = 0;
300                 memcpy(m_buffer, &chdr, sizeof(CRYPT_HEADER)
                        );
301                 m_rawbytes += nbytes;
302                 nbytes += sizeof(CRYPT_HEADER);
303             }
304             // benchmark
305             /*
306             gettimeofday(&tp, NULL);
307             t2 = (double)tp.tv_sec + (1.e-6) * tp.tv_usec;
308             snprintf(benchbuf, sizeof(benchbuf), "%f\n", (t2
                    -t1));
309             fwrite(benchbuf, 1, strlen(benchbuf), fd);
310             */
```

E.2 Multicast-dekryptering (klient)

Dekryptering av multicast på klientmodulen (`vldrm:184-409`).

```
184 bool client_init(){
185     INFO("starting services");
186 
187     MediaClient client(g_config.getHttpPort());
188     if ( !client.bind() )
189         return false;
190     ClientSession tls;
191 
192     CBString request;
193     CBString reqproto;
194     MediaClientSession* session;
195     Reader* reader;
196     UdpReader* udp;
197     int fd, mcast_port;
198     size_t nbytes;
199     unsigned char buffer[NETBUFSIZE];
200     unsigned char* buf;
201 
202     // benchmark
203     /*
204     double t1, t2;
205     struct timeval tp;
206     char benchbuf[1024];
207     FILE* fp = fopen("client_bench.out", "w");
208     */
209 
210     for (;;){
211         fd = client.accept();
212 
213         if ( g_exit_pending || g_restart_pending )
214             break;
215 
216         if ( fd < 1 )
217             continue;
218 
219         session = new MediaClientSession(fd, &client);
```

```
220         if ( !session->run(request, mcast_port) ){
221             delete session;
222             continue;
223         }
224 
225         if ( session->isMulticast() ){
226             // open and join multicast stream
227             mcast_port = (mcast_port == 0 ? 1234 :
                    mcast_port);
228             reader = new UdpReader(request, mcast_port);
229             udp = (UdpReader*)reader;
230             udp->open(2);
231 
232             if ( strncmp(request, "232.", 4) == 0 ){
233                 // TODO
234                 // SSM source address should be more dynamic
                        ,
235                 // eg. the server could transmit the source
                        of the
236                 // mutlicast through the TLS connection.
237                 DEBUGP("joining SSM multicast channel: %s/%s
                        ",
238                         (const char*)g_config.getDrmServer()
                            ,
239                         (const char*)request);
240                 udp->join(g_config.getDrmServer()); // SSM
241             }else{
242                 DEBUGP("joining ASM multicast group: %s",
243                         (const char*)request);
244                 udp->join(); // ASM
245             }
246 
247             // check that the requested multicast stream is
                    actually present
248             nbytes = reader->read(buffer, NETBUFSIZE);
249             if ( nbytes < 1 ){
250                 ERROR("multicast stream not responding");
251                 udp->leave();
252                 delete reader;
253                 session->sendStatus(404);
254                 delete session;
255                 continue;
256             }else{
257                 session->sendHeader(200, 0, "application/
                        octet-stream");
258             }
259 
260             reqproto.format("MULTICAST/%s:%d",
261                     (const char*)request, mcast_port);
262         }else{
263             // open TCP connection to vldrmd
264             reqproto = request;
265             ERROR("NOT SUPPORTED YET");
266             session->sendStatus(404);
267             delete session;
268             continue;
269         }
270 
271         // key is passed to the cipher object and deleted by
```

```
             its destructor
272          CipherKey* key = new CipherKey(Cipher::CIPHER_NONE,
                 Cipher::MODE_NONE);
273          if ( !tls.getKey(reqproto, key) ){
274              ERROR("Access denied to %s", (const char*)
                     reqproto);
275              session->sendStatus(403);
276              delete session;
277              continue;
278          }
279          Cipher cipher(key);
280
281          unsigned int ivsize = 8;
282          unsigned int blocksize = cipher.getBlocksize();
283          unsigned char* iv = new unsigned char[blocksize];
284          CRYPT_HEADER chdr;
285
286          // recieve, decrypt and forward
287          // This code expects CTR mode
288          int msgdiff;
289          unsigned int lastmsg = 0;
290          unsigned int keyid = key->getID();
291
292          unsigned char k[21];
293          key->getKey(k, 20);
294          k[20] = 0;
295
296          do{
297              if ( cipher.getCipher() != Cipher::CIPHER_NONE )
298                  tls.mCastKeyPoll(key);
299
300              if ( (nbytes = reader->read(buffer, NETBUFSIZE))
                      <= 0 ){
301                  WARN("MULTICAST READ FAILED");
302                  break;
303              }
304
305              // benchmark
306              /*
307              gettimeofday(&tp, NULL);
308              t1 = (double)tp.tv_sec + (1.e-6) * tp.tv_usec;
309              */
310
311              // package must be larger than the header (
                     static)
312              if ( nbytes < (sizeof(CRYPT_HEADER)) ){
313                  WARN("NOT ENOUGH DATA FOR HEADER");
314                  continue;
315              }
316
317              memcpy(&chdr, buffer, sizeof(CRYPT_HEADER));
318
319              if ( lastmsg > 0 ){
320                  msgdiff = chdr.msgId - lastmsg;
321                  // UDP messages out of order or lost
322                  if ( msgdiff > 1 ){
323                      WARN("UDP MESSAGES LOST msgId(%08x)
                             lastmsg(%08x) diff(%08x)",
324                                  chdr.msgId, lastmsg, msgdiff);
```

```
325                 }else if ( msgdiff < 1 && msgdiff > -512 ){
326                     WARN("UDP MESSAGE OUT OF ORDER msgId(%08
                            x) lastmsg(%08x) diff(%d)",
327                             chdr.msgId, lastmsg, msgdiff);
328                     lastmsg = chdr.msgId;
329                     continue;
330                 }
331             }
332             lastmsg = chdr.msgId;
333 
334             if ( chdr.keyId != 0 ){
335                 // package must be larger than the header (
                        variable)
336                 if ( nbytes < (sizeof(CRYPT_HEADER) + ivsize
                        ) ){
337                     WARN("NOT ENOUGHT DATA FOR HEADER 2");
338                     continue;
339                 }
340             }
341 
342             if ( chdr.keyId == 0 )
343                 buf = buffer + sizeof(CRYPT_HEADER);
344             else
345                 buf = buffer + sizeof(CRYPT_HEADER) + ivsize
                        ;
346 
347             if ( cipher.getCipher() != Cipher::CIPHER_NONE
                    && chdr.keyId != 0 ){
348                 if ( keyid < chdr.keyId && key->getNextID()
                        > 0 ){
349                     if ( key->getNextID() != chdr.keyId ){
350                         DEBUGP("KEY SWAP KEY MISMATCH");
351                         break; // encryption will fail, bail
                                out
352                     }
353                     if ( !key->swapKeys() ){
354                         DEBUGP("KEY SWAP FAILED");
355                         break; // encryption will fail, bail
                                out
356                     }
357                     if ( !cipher.activateKey() ){
358                         DEBUGP("KEY ACTIVATE FAILED");
359                         break;
360                     }
361                     keyid = key->getID();
362                     DEBUGP("USING KEY %d", keyid);
363                 }
364                 if ( keyid != chdr.keyId )
365                     WARN("KEY ID NOT MATCHED WITH HEADER key
                            (%d) hdr(%d)",
366                             keyid, chdr.keyId);
367 
368                 // Recreate the IV
369                 memset(iv, 0, blocksize);
370                 key->getNonce(iv, key->getNoncelen()); //
                        Nonce
371                 memcpy(iv + key->getNoncelen(),
372                         buffer + sizeof(CRYPT_HEADER),
                            ivsize); // IV
```

```
373                     iv[blocksize-1] = 0x01;          // counter =
                            1
374                     cipher.setIV(iv, blocksize);
375
376                     cipher.decrypt(buf, nbytes, NULL, 0);
377                     nbytes -= sizeof(CRYPT_HEADER) + ivsize;
378                 }else{
379                     nbytes -= sizeof(CRYPT_HEADER);
380                 }
381
382                 // benchmark
383                 /*
384                 gettimeofday(&tp, NULL);
385                 t2 = (double)tp.tv_sec + (1.e-6) * tp.tv_usec;
386                 snprintf(benchbuf, sizeof(benchbuf), "%f\n", (t2
                        -t1));
387                 fwrite(benchbuf, 1, strlen(benchbuf), fp);
388                 */
389
390                 if ( session->send(buf, nbytes) < 0 ){
391                     if ( errno != ECONNRESET )
392                         WARN("HTTP SEND FAILED: %s", strerror(
                                errno));
393                     break;
394                 }
395
396             } while ( session->isConnected() &&
397                     (!g_exit_pending && !g_restart_pending) );
398
399             tls.disconnect();
400             delete [] iv;
401             delete reader;
402             delete session;
403
404             if ( g_exit_pending || g_restart_pending )
405                 break;
406         }
407
408         return true;
409     }
```

E.3 Nøkkelbytte

Utvalgte metoder fra den asynkrone nøkkelbyttetråden (`mcastkeychanger.cpp:49-79` og `tls_session.cpp:468-518`).

```
49  void McastKeyChanger::run(){
50      DEBUGP("Starting thread to feed mcast key");
51      unsigned int mcastid=0;
52      unsigned int mcastid2=0;
53      for(;;)
54      {
55          Thread::sleep(2000);
56          // check every mcast thread
57          for (mcastid = 0; mcastid < g_mcast.size(); mcastid
                ++)
58          {
```

```
59              if ( g_mcast[mcastid]  && g_mcast[mcastid]->
                    changeKey())
60              {
61                   g_mcast[mcastid]->getKey(m_key);
62                   // notify all connected clients
63                   for(unsigned int i = 0; i < m_tlsServer->
                        sessions.size();i++)
64                   {
65                        if(m_tlsServer->sessions[i]->
                             getMulticast(mcastid2))
66                        {
67                             if(mcastid2==mcastid)
68                             {
69                                  m_tlsServer->sessions[i]->
                                       changeMcastKey(m_key);
70                             }
71                        }
72                   }
73                   // need a small delay here for the client to
                         recieve the key
74                   Thread::sleep(1000);
75                   g_mcast[mcastid]->readyKey(); // notify
                        mcast thread
76              }
77         }
78     }
79 }
```

```
468 void TlsSession::changeMcastKey(CipherKey& key)
469 {
470     m_Key = key;
471     memset(m_buffer,0,MAX_BUF); ///<Setting everything in
            the buffer to 0
472     char cbuf[20];
473     char mbuf[20];
474     char* encodedcurrent = new char[(m_Key.getKeylen() +
            m_Key.getNoncelen()+2)/3*4+1];
475     unsigned char* unencodedCurrent = new unsigned char[
            m_Key.getKeylen() + m_Key.getNoncelen()];
476
477     Cipher::mapName(m_Key.getCipher(), cbuf, 20);
478     Cipher::mapMode(m_Key.getMode(), mbuf, 20);
479
480     m_Key.getKey(unencodedCurrent,
481             m_Key.getKeylen() + m_Key.getNoncelen(),
482             false);
483     //b64 encode the key
484     b64Encode(unencodedCurrent, m_Key.getKeylen() + m_Key.
            getNoncelen(),
485             encodedcurrent,
486             (m_Key.getKeylen() + m_Key.getNoncelen()+2)
                    /3*4+1);
487
488
489     char* encodednext = new char[(m_Key.getKeylen() + m_Key.
            getNoncelen()+2)/3*4+1];
490     unsigned char* unencodednext = new unsigned char[m_Key.
            getKeylen() + m_Key.getNoncelen()];
491     m_Key.getKey(unencodednext,
```

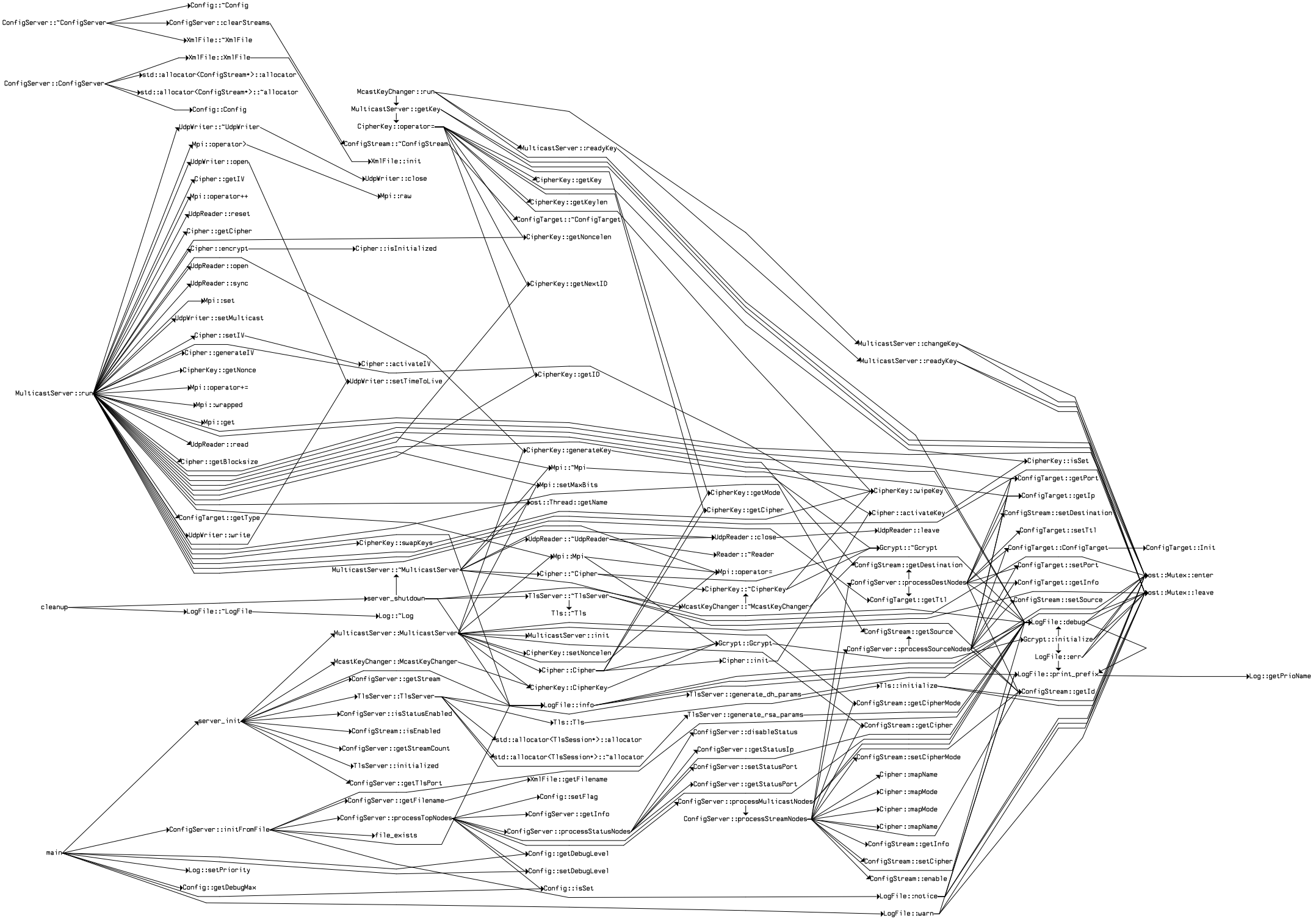
```
492             m_Key.getKeylen() + m_Key.getNoncelen(),
493             true);
494
495     b64Encode(unencodednext, m_Key.getKeylen() + m_Key.
            getNoncelen(),
496             encodednext,
497             (m_Key.getKeylen() + m_Key.getNoncelen()+2)
                    /3*4+1);
498     snprintf(m_buffer, MAX_BUF,
499             "KEY %s %s %d %d %d %s %d %d %d %s",
500             cbuf,
501             mbuf,
502             m_Key.getID(),
503             m_Key.getKeylen(),
504             m_Key.getNoncelen(),
505             encodedcurrent,
506             m_Key.getNextID(),
507             m_Key.getKeylen(),
508             m_Key.getNoncelen(),
509             encodednext);
510
511     delete[] encodednext;
512     delete[] unencodednext;
513     delete[] encodedcurrent;
514     delete[] unencodedCurrent;
515
516     gnutls_record_send(m_session, m_buffer, strlen(m_buffer)
            );
517     NOTICE("TLS: sent new multicast key to %s", (const char
            *)m_userID);
518 }
```

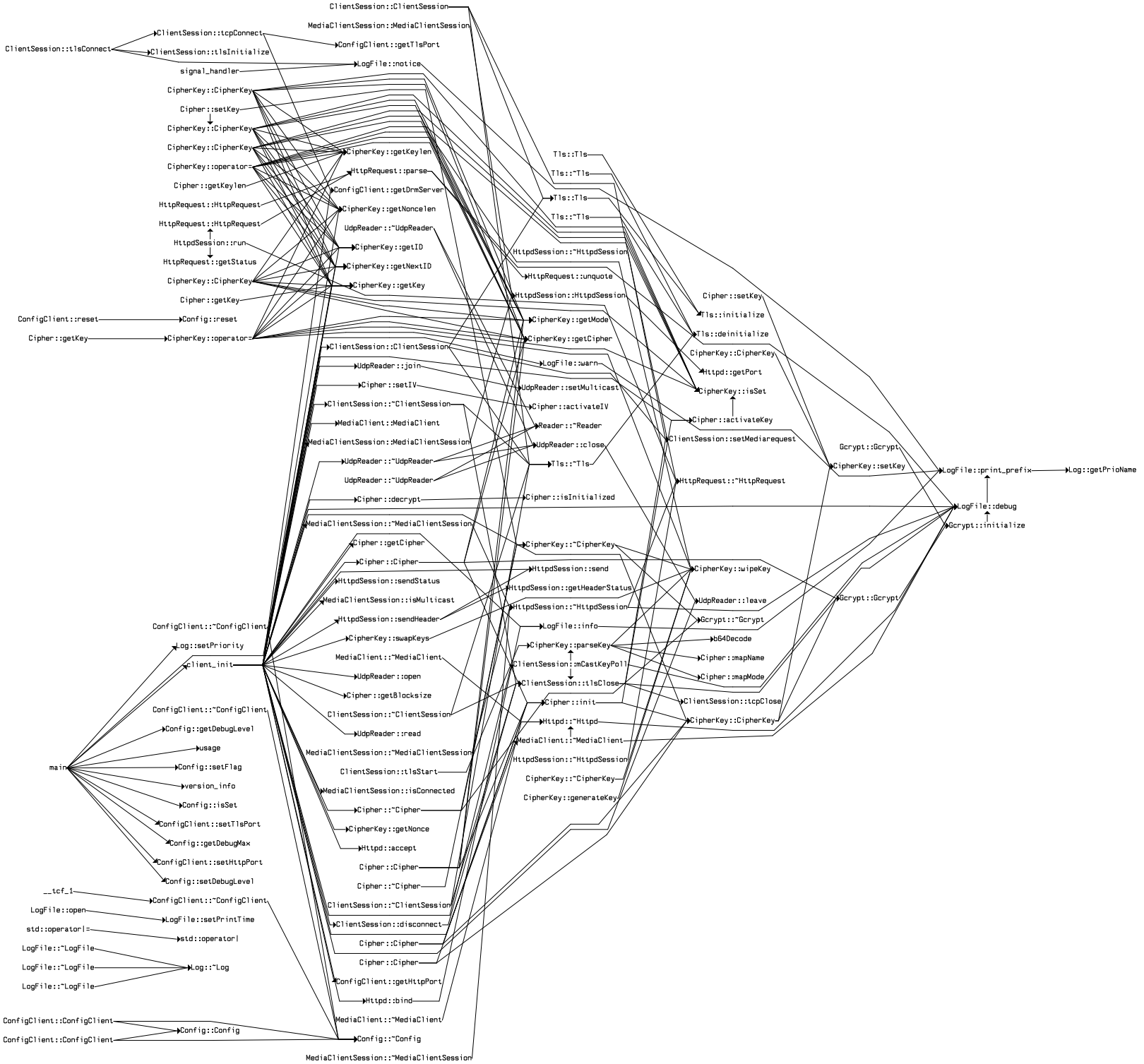
Tillegg F

Funksjonskallgraf

De to neste sidene viser funksjonskallgrafer for henholdsvis servermodul og klientmodul. Metoden som er brukt for å generere disse grafene inkluderer eksekveringsprofilen fra *gprof* (se avsnitt 7.2.5). Fremgangsmetoden for å generere grafen er beskrevet på følgende nettsted: http://www.ida.liu.se/ vaden/cgdi/

Grafene inneholder mye informasjon for en A4-side. En PDF-versjon av dette dokumentet egner seg bedre (se vedlegg G), hvor man kan forstørre interessante områder. Se avsnitt 7.2.5 for en beskrivelse av *gprof* og funksjonsgrafene.


ConfigServer::~ConfigServer
Config::~Config
ConfigServer::clearStreams
XmlFile::~XmlFile
ConfigServer::ConfigServer
XmlFile::XmlFile
std::allocator<ConfigStream*>::allocator
std::allocator<ConfigStream*>::~allocator
Config::Config
ConfigStream::~ConfigStream
XmlFile::init
McastKeyChanger::run
MulticastServer::getKey
CipherKey::operator=
MulticastServer::readyKey
CipherKey::getKey
CipherKey::getKeylen
ConfigTarget::~ConfigTarget
CipherKey::getNoncelen
CipherKey::getNextID
UdpWriter::~UdpWriter
Mpi::operator>
UdpWriter::open
Cipher::getIV
Mpi::operator++
UdpReader::reset
Cipher::getCipher
Cipher::encrypt
Cipher::isInitialized
UdpReader::open
UdpReader::sync
Mpi::set
UdpWriter::setMulticast
Cipher::setIV
Cipher::generateIV
CipherKey::getNonce
Mpi::operator+=
Mpi::wrapped
Mpi::get
UdpReader::read
Cipher::getBlocksize
ConfigTarget::getType
UdpWriter::write
UdpWriter::close
Mpi::raw
Cipher::activateIV
UdpWriter::setTimeToLive
CipherKey::getID
CipherKey::swapKeys
MulticastServer::run
MulticastServer::changeKey
MulticastServer::readyKey
CipherKey::generateKey
Mpi::~Mpi
Mpi::setMaxBits
Ost::Thread::getName
CipherKey::getMode
CipherKey::getCipher
CipherKey::wipeKey
Cipher::activateKey
UdpReader::~UdpReader
UdpReader::close
Reader::~Reader
Mpi::Mpi
Cipher::~Cipher
CipherKey::~CipherKey
MulticastServer::~MulticastServer
UdpReader::leave
Gcrypt::~Gcrypt
ConfigStream::getDestination
ConfigServer::processDestNodes
ConfigTarget::getTtl
server_shutdown
TlsServer::~TlsServer
Tls::~Tls
McastKeyChanger::~McastKeyChanger
cleanup
LogFile::~LogFile
Log::~Log
MulticastServer::MulticastServer
MulticastServer::init
CipherKey::setNoncelen
Cipher::Cipher
CipherKey::CipherKey
Gcrypt::Gcrypt
Cipher::init
McastKeyChanger::McastKeyChanger
ConfigServer::getStream
TlsServer::TlsServer
ConfigServer::isStatusEnabled
ConfigStream::isEnabled
ConfigServer::getStreamCount
TlsServer::initialized
ConfigServer::getTlsPort
LogFile::info
Tls::Tls
std::allocator<TlsSession*>::allocator
std::allocator<TlsSession*>::~allocator
TlsServer::generate_dh_params
TlsServer::generate_rsa_params
server_init
ConfigServer::initFromFile
ConfigServer::getFilename
ConfigServer::processTopNodes
file_exists
XmlFile::getFilename
Config::setFlag
ConfigServer::getInfo
ConfigServer::processStatusNodes
ConfigServer::disableStatus
ConfigServer::getStatusIp
ConfigServer::setStatusPort
ConfigServer::getStatusPort
ConfigServer::processMulticastNodes
ConfigServer::processStreamNodes
main
Log::setPriority
Config::getDebugMax
Config::getDebugLevel
Config::setDebugLevel
Config::isSet
Tls::initialize
ConfigStream::getCipherMode
ConfigStream::getCipher
ConfigStream::setCipherMode
Cipher::mapName
Cipher::mapMode
Cipher::mapMode
Cipher::mapName
ConfigStream::getInfo
ConfigStream::setCipher
ConfigStream::enable
LogFile::notice
LogFile::warn
CipherKey::isSet
ConfigTarget::getPort
ConfigTarget::getIp
ConfigStream::setDestination
ConfigTarget::setTtl
ConfigTarget::ConfigTarget
ConfigTarget::Init
ConfigTarget::setPort
ConfigTarget::getInfo
ConfigStream::setSource
Ost::Mutex::enter
Ost::Mutex::leave
LogFile::debug
Gcrypt::initialize
LogFile::err
LogFile::print_prefix
Log::getPrioName
ConfigStream::getId
ConfigStream::getSource
ConfigServer::processSourceNodes


ClientSession::ClientSession
MediaClientSession::MediaClientSession
ClientSession::tcpConnect
ClientSession::tlsConnect
ClientSession::tlsInitialize
ConfigClient::getTlsPort
signal_handler
LogFile::notice
CipherKey::CipherKey
Cipher::setKey
CipherKey::CipherKey
CipherKey::CipherKey
CipherKey::getKeylen
CipherKey::operator=
HttpRequest::parse
Cipher::getKeylen
ConfigClient::getDrmServer
HttpRequest::HttpRequest
CipherKey::getNonceLen
HttpRequest::HttpRequest
UdpReader::~UdpReader
HttpdSession::run
CipherKey::getID
HttpRequest::getStatus
CipherKey::getNextID
CipherKey::CipherKey
CipherKey::getKey
Cipher::getKey
ConfigClient::reset
Config::reset
Cipher::getKey
CipherKey::operator=
Tls::Tls
Tls::~Tls
Tls::Tls
Tls::~Tls
HttpdSession::~HttpdSession
HttpRequest::unquote
HttpdSession::HttpdSession
CipherKey::getMode
CipherKey::getCipher
ClientSession::ClientSession
LogFile::warn
UdpReader::join
UdpReader::setMulticast
Cipher::setIV
Cipher::activateIV
ClientSession::~ClientSession
Reader::~Reader
MediaClient::MediaClient
UdpReader::close
MediaClientSession::MediaClientSession
Tls::~Tls
UdpReader::~UdpReader
UdpReader::~UdpReader
Cipher::decrypt
Cipher::isInitialized
MediaClientSession::~MediaClientSession
Cipher::getCipher
CipherKey::~CipherKey
Cipher::Cipher
HttpdSession::send
HttpdSession::sendStatus
HttpdSession::getHeaderStatus
MediaClientSession::isMulticast
HttpdSession::~HttpdSession
HttpdSession::sendHeader
LogFile::info
ConfigClient::~ConfigClient
CipherKey::swapKeys
CipherKey::parseKey
Log::setPriority
MediaClient::~MediaClient
ClientSession::mCastKeyPoll
client_init
UdpReader::open
ClientSession::tlsClose
Cipher::getBlocksize
Cipher::init
ConfigClient::~ConfigClient
ClientSession::~ClientSession
Httpd::~Httpd
UdpReader::read
MediaClient::~MediaClient
MediaClientSession::~MediaClientSession
HttpdSession::~HttpdSession
main
ClientSession::tlsStart
CipherKey::~CipherKey
MediaClientSession::isConnected
CipherKey::generateKey
Cipher::~Cipher
CipherKey::getNonce
Httpd::accept
Cipher::Cipher
Cipher::~Cipher
ConfigClient::~ConfigClient
Config::getDebugLevel
usage
Config::setFlag
version_info
Config::isSet
ConfigClient::setTlsPort
Config::getDebugMax
ConfigClient::setHttpPort
Config::setDebugLevel
__tcf_1
ConfigClient::~ConfigClient
LogFile::open
LogFile::setPrintTime
std::operator|=
std::operator|
LogFile::~LogFile
Log::~Log
LogFile::~LogFile
LogFile::~LogFile
ClientSession::~ClientSession
ClientSession::disconnect
Cipher::Cipher
Cipher::Cipher
ConfigClient::getHttpPort
Httpd::bind
MediaClient::~MediaClient
ConfigClient::ConfigClient
Config::Config
ConfigClient::ConfigClient
Config::~Config
MediaClientSession::~MediaClientSession
Cipher::setKey
Tls::initialize
Tls::deinitialize
CipherKey::CipherKey
Httpd::getPort
CipherKey::isSet
Cipher::activateKey
ClientSession::setMediaRequest
HttpRequest::~HttpRequest
CipherKey::wipeKey
UdpReader::leave
Gcrypt::~Gcrypt
b64Decode
Cipher::mapName
Cipher::mapMode
ClientSession::tcpClose
CipherKey::CipherKey
Gcrypt::Gcrypt
CipherKey::setKey
LogFile::print_prefix
Log::getPrioName
LogFile::debug
Gcrypt::initialize
Gcrypt::Gcrypt

Tillegg G

CD-ROM/Kildekode

Vedlagt følger en CD-ROM[1] som inneholder all kildekode for prototypen som er utviklet som en del av oppgaven. Se vedlegg B for en beskrivelse av hvordan kildekoden kompileres og kjøres.

Tabell G.1 gir en beskrivelse av innholdet på CD-ROMen.

Filnavn	Beskrivelse
codemetrics	Resultat av kodeanalyse generert av CCCC. Se `index.html`.
dokumentasjon	Katalog med API-dokumentasjon generert av Doxygen. Se `index.html`.
kildekode	Katalog med kildekoden til systemet.
vldrm-0.1.tar.gz	Uttrekk av kildekoden fra CVS.
vldrm-0.1.zip	Samme innhold som filen ovenfor, men i ZIP-format.
vldrm-docs.tar.gz	API-dokumentasjon generert av Doxygen.
vldrm-docs.zip	Samme innhold som filen ovenfor, men i ZIP-format.

Tabell G.1: Innhold på CD-ROM

[1] Innholdet av CD-ROMen er også tilgjengelig på Internett på følgende sider: http://www.stud.ntnu.no/groups/drm/ og http://phuzz.org/download/ntnu/. NTNU-siden vil mest sannsynlig ikke være tilgjengelig etter vi har avsluttet våre studier.

Bibliografi

[AAMS01] Z. Albanna, K. Almeroth, D. Meyer og M. Schipper. *RFC3171. IANA Guidelines for IPv4 Multicast Address Assignments.* The Internet Society/Juniper Networks/UCSB/Sprint/IANA, 2001.

[All04] Internet Streaming Media Alliance. Encryption and authentication specification. 26, 2 2004.

[AM04] Christian Andersen og Lars Moe. Digital rettighetsstyring, fordypningprosjekt. Rapport, NTNU, 2004.

[BdWH+03] Ian Burnett, Rik Van de Walle, Keith Hill, Jan Bormasn og Fernando Pereira. Mpeg-21: Goals and achievements. *IEEE Multimedia*, 10 2003.

[BH] Jan Bormans og Keith Hill. Mpeg-21 overview v.5. http://www.chiariglione.org/mpeg/standards/mpeg-21/mpeg-21.htm, [17.03.2005].

[Bha03] S. Bhattacharyya. *RFC3569. An Overview of Source-Specific Multicast (SSM).* The Internet Society/Sprint, 2003.

[BMN+04] M. Baugher, D. McGrew, M. Naslund, E. Carrara og K. Norrman. *RFC3711. The Secure Real-time Transport Protocol (SRTP).* The Internet Society/Cisco Systems/Ericsson Research, 2004.

[Bøh04] Jon Bøhmer. Snap tv, development road-map. 10 2004.

[CC] Inc Cunningham & Cunningham. What is refactoring. http://c2.com/cgi/wiki?WhatIsRefactoring, [09.06.2005].

[Con] ContentGuard. Mpeg-21 overview v.5. http://www.contentguard.com/MPEGREL_home.asp, [28.05.2005].

[Cro02] Michael Cross. *Security+ Study Guide and DVD Training System.* Syngress Publishing, 2002. http://site.ebrary.com/lib/ntnu/Doc?id=10021818. ISBN 1931836-728.

[dG] Juan-Mariano de Goyeneche. Multicast over tcp/ip howto. http://www.tldp.org/HOWTO/Multicast-HOWTO.html, [25.05.2005].

[Dwo01] Morris Dworkin. Recommendation for block cipher modes of operation. *NIST Special Publication 800-38A*, 2001.

[Fai] Dr. Gorry Fairhurst. Mpeg-2 transmission. http://erg.abdn.ac.uk/research/future-net/digital-video/mpeg2-trans.html, [25.05.2005].

[FK04] Borko Furht og Darko Kirovski. *Multimedia Security Handbook.* CRC Press, 2004. ISBN 0-8493-2773-3.

[FS03] Niels Ferguson og Bruce Schneier. *Practical Cryptography.* Wiley Publishing, Inc., 2003. ISBN 0-471-22357-3.

[GMDS98] Carsten Griwoz, Oliver Merkel, Jana Dittmann og Ralf Steinmetz. Protecting vod the easier way. *ACM*, side 21–28, 1998.

[Gro] Moving Picture Experts Group. Mpeg - frequently asked questions. http://www.chiariglione.org/mpeg/faq.htm, [21.04.2005].

[HD03] Thomas Hardjono og Lakshminath R. Dondeti. *Multicast and Group Security.* Artech house, Inc, 2003. ISBN 1-58053-342-6.

[Hou99] R. Housley. *RFC2630. Cryptographic Message Syntax.* The Internet Society/SPYRUS, 1999.

[Hou04] R. Housley. *RFC3686. Using Advanced Encryption Standard (AES) Counter Mode With IPsec Encapsulating Security Payload (ESP).* The Internet Society/Vigil Security, 2004.

[ID] Network Working Group Internet-Draft. Ssh protocol architecture. http://www.snailbook.com/docs/architecture.txt, [07.03.2005].

[LSKG03] Tom Lokkabaugh, Dougkas C. Sicker, David M. Keaton og Wang Ye Guo. Security analysis of selectively encrypted mpeg-2 streams. 09 2003. http://spot.colorado.edu/ lookabau/Documents/Securityctively

[PD03] Larry L. Peterson og Bruce S. Davie. *Computer Networks*. Elsevier, 2003. ISBN 1-55860-833-8.

[PE02] Fernando Pereira og Touradj Ebrahimi. *The MPEG-4 Book*. IMSC Press, 2002. ISBN 0-13-061621-4.

[Per03] Colin Perkins. *RTP - Audio and Video for the Internet*. Addison-Wesley, 2003. ISBN 0-672-32249-8.

[PW04] HweeHwa Pang og Yougdong Wu. Evaluation of mpeg-4 ipmp extension. *Institute for Infocomm Research*, 2004.

[SB98] Changgui Shi og Bharat K. Bhargava. A fast mpeg video encryption algorithm. I *ACM Multimedia*, side 81–88, 1998.

[Sch96] Bruce Schneier. *Applied Cryptography, 2nd edt.* John Wiley & Sons, Inc, 1996. ISBN 0-471-12845-7.

[Sch04] Edward M. Schwalb. *iTV Handbook*. Bernard Goodwin, 2 utgave, 2004. ISBN 0-13-100312-7.

[Sof] Quality Engineering Software. Manual and automated software testing. http://www.qestest.com/principl.htm, [02.06.2005].

[Sta03] William Stallings. *Network security essentials*. Prentice Hall, 2003. ISBN 0-13-120271-5.

[STM04] STMicroelectronics. Product preview stb7109. low-cost hdtv set-top box decoder chip. 04 2004.

[Tan96] Lei Tang. Methods for encrypting and decrypting mpeg video data efficiently. *Carnegie Mellon University/ACM*, 1996.

[Wat04] John Watkinson. *The MPEG handbook*. Elsevier, 2 utgave, 2004. ISBN 0-240-80578-X.

[Wika] Wikipedia. Iptv. http://en.wikipedia.org/wiki/Iptv, [14.02.2005].

[Wikb] Wikipedia. Trusted computing. http://en.wikipedia.org/wiki/Trusted_computing, [09.06.2005].

[Wikc] Wikipedia. Unit testing. http://en.wikipedia.org/wiki/Unit_testing, [07.02.2005].

[Wikd] Wikipedia. Vod. http://en.wikipedia.org/wiki/Video_on_demand, [14.02.2005].

[Wike] Wikipedia. X.509. http://en.wikipedia.org/wiki/X.509, [10.02.2005].

[WN94] David J. Wheeler og Roger M. Needham. Tea, a tiny encryption algorithm. *Cambridge University, England*, 1994.

[WN97] David J. Wheeler og Roger M. Needham. Tea extension. *Cambridge University, England*, 1997.

[WRSR04] Bill Fenner W. Richard Stevens og Andrew M. Rudoff. *UNIX Network Programming, The sockets networking API.* Addison-Wesly, 3 utgave, 2004. ISBN 0-13-141155-1.